Oracles du test de transformations de modèles

Olivier Finot

UNIVERSITÉ DE NANTES

UFR DES SCIENCES ET TECHNIQUES

SCIENCES ET TECHNOLOGIES
DE L'INFORMATION ET DE MATHÉMATIQUES (STIM)

Année 2014

N° attribué par la bibliothèque

Oracles du test de transformations de modèles

THÈSE DE DOCTORAT
Discipline : Informatique

Présentée
et soutenue publiquement par

Olivier FINOT

le 3 février 2014 au LINA, devant le jury ci-dessous

Président	:	Antoine BEUGNARD, Professeur HDR	TELECOM Bretagne
Rapporteurs	:	Slimane HAMMOUDI, Maître de conférences, HDR	ESEO Angers
		Christian PERCEBOIS, Professeur d'Université	Université de Toulouse

Directeur de thèse : Christian ATTIOGBE
Co-encadrant de thèse : Jean-Marie MOTTU
Co-encadrant de thèse : Gerson SUNYÉ
Laboratoire : **L**ABORATOIRE D'**I**NFORMATIQUE DE **N**ANTES **A**TLANTIQUE.
2, rue de la Houssinière, BP 92 208 – 44 322 Nantes, CEDEX 3.

ORACLES DU TEST DE TRANSFORMATIONS DE MODÈLES

Model Transformation Test Oracles

Olivier FINOT

⋈

favet neptunus eunti

Université de Nantes

Olivier FINOT

Oracles du test de transformations de modèles

xi+133 p.

Ce document a été préparé avec LaTeX2e et la classe `these-LINA` version v. 1.30 de l'association de jeunes chercheurs en informatique LoGiN, Université de Nantes. La classe `these-LINA` est disponible à l'adresse :

`http://login.univ-nantes.fr/`

Impression : these_OFINOT.tex – 13/2/2014 – 10:50

Révision pour la classe : these-LINA.cls,v 1.30 2005/08/16 17:01:41 mancheron Exp

Résumé

L'Ingénierie Dirigée par les Modèles place les modèles au cœur du cycle de développement logiciel. Ces modèles évoluent par le biais de diverses transformations. Dans cette thèse nous nous sommes intéressés à la validation de ces transformations de modèles par le test, et en particulier à l'oracle de ce test. Nous proposons deux approches pour assister le testeur dans la création de ces oracles. Tout d'abord, nous offrons une assistance passive en fournissant au testeur une nouvelle fonction d'oracle. Cette dernière lui permet de créer des oracles qui ne contrôlent qu'une partie des modèles obtenus. Nous avons défini la notion de verdict partiel, explicité les situations où un verdict partiel est plus avantageux et proposé un protocole global du test de transformations dans ce contexte. Nous avons mis en œuvre cette première proposition dans un outillage avec lequel nous l'avons expérimentée. Ensuite, nous offrons au testeur une assistance active en étudiant la qualité d'un ensemble d'oracles. Nous considérons la qualité d'un ensemble d'oracles selon sa capacité à détecter des fautes dans la transformation sous test. Nous proposons une méthode qui corrige en partie les insuffisances de l'analyse de mutation, utilisée dans ce contexte ; nous mesurons la couverture du méta-modèle de sortie par l'ensemble d'oracles considéré. Nous montrons que notre approche est indépendante du langage utilisé pour la mise en œuvre de la transformation sous test, et fournit au testeur des informations pour l'amélioration des oracles. Nous avons défini une démarche pour mesurer la couverture et qualifier des oracles. Nous avons développé un outil pour expérimenter et valider notre proposition.

Mots-clés : Ingénierie Dirigée par les Modèles, Transformations de modèles, Oracle, Qualification

Abstract

With Model Driven Engineering models are the heart of software development. These models evolve through transformations. In this thesis our interest was the validation for these model transformations by testing, and more precisely the test oracles. We propose two approaches to assist the tester to create these oracles. With the first approach this assistance is passive; we provide the tester with a new oracle function. The test oracles created with this new oracle function control only part of the model produced by the transformation under test. We defined the notion of partial verdict, described the situations where having a partial verdict is beneficial for the tester and how to test a transformation in this context. We developed a tool implementing this proposal, and ran experiments with it. With the second approach, we provide a more active assistance about test oracles' quality. We study the quality of a set of model transformation test oracles. We consider that the quality of a set of oracles is linked to its ability to detect faults in the transformation under test. We show the limits of mutation analysis which is used for this purpose, then we propose a new approach that corrects part of these drawbacks. We measure the coverage of the output meta-model by the set of oracles we consider. Our approach does not depend on the language used for the transformation under test's implementation. It also provides the tester with hints on how to improve her oracles. We defined a process to evaluate meta-model coverage and qualify test oracles. We developed a tool implementing our approach to validate it through experimentations.

Keywords: Model Driven Engineering, Model Transformations, Oracle, Qualification

Remerciements

Avant toute chose, je tiens à remercier mes encadrants pour m'avoir permis de découvrir le monde de la recherche. Durant ces années de thèse il n'ont eu de cesse de me pousser à avancer ; ce qui n'a pas dû être toujours facile. Merci à Jean-Marie MOTTU, ses encouragements et les discussions avec lui m'ont permis de mener cette thèse à son terme. Je remercie également Gerson SUNYÉ qui lui aussi a toujours été présent lorsque j'ai eu besoin de lui. Merci aussi à Chrisrian ATTIOGBE pour avoir dirigé cette thèse et pour ses conseils toujours avisés.

Je remercie tous les membres du jury qui ont accepté d'évaluer mes travaux. Merci particulièrement à Antoine BEUGNARD et Christian PERCEBOIS qui on suivi cette thèse dès la première année en tant que membres de mon comité de suivi. Leurs questions et commentaires pertinents tout au long de ces trois ans m'ont été très utiles pour avancer. Merci aussi à Slimane HAMMOUDI d'avoir rapporté cette thèse, ses observations ont elles aussi été des plus utiles.

Merci également à tous les membres de l'équipe AeLoS pour leur accueil. Dès mon arrivée dans ce laboratoire où je n'avais aucun repère, ils m'ont fait me sentir à l'aise. Les discussions partagées avec eux ont été des plus enrichissantes. De la même manière, je remercie tous les personnels du LINA qui ont aidé au bon déroulement de cette thèse.

Merci aussi à tous les doctorants que j'ai eu le plaisir de rencontrer durant ces trois années au sein du LINA et de l'association LOGIN ; merci pour leur soutient et leur amitié. Je les remercie également pour tous les échanges qui ont eu lieu lors des DHDs ou en d'autres occasions. Ces échanges avec des personnes qui ne s'intéressent pas forcément au même domaine ont toujours été très intéressants et m'ont aidé à prendre du recul sur mon propre travail.

Je n'oublie pas non plus mes amis qui ne m'ont jamais lâché, même dans les moments les plus difficiles où je n'ai pas été très présent pour eux. Ils m'ont toujours encouragé tout au long de cette thèse et je les en remercie. Merci en particulier à Frédéric et Nathalie qui ont prit le temps de relire ce manuscrit tout en étant très occupés à coté de cela.

Enfin, je remercie ma famille. Eux aussi n'ont eu de cesse de m'encourager et ont toujours été derrière moi, à me pousser en cas de besoin. Ils ont également sus se montrer compréhensifs lorsqu'à cause de la charge de travail je n'ai pu les voir aussi souvent qu'ils l'auraient souhaité, et moi aussi.

Table des matières

CHAPITRE 1

Introduction

1.1 Contexte et problématique

L'informatique a une importance grandissante dans notre vie. Tous nos équipements tendent à être de plus en plus "intelligents". De nombreux logiciels fonctionnent sur ces équipements. Il est important de s'assurer de la correction de ces logiciels par rapport à leur spécification, pour qu'ils puissent être utilisés en toute confiance. C'est là qu'intervient le génie logiciel, l'application d'une approche systématique, disciplinée, et mesurable à la conception, au développement, à l'exploitation, à la maintenance, et à l'évaluation de logiciels.

Le test logiciel est une des activités du génie logiciel qui vise à vérifier et à valider des logiciels. Tester un programme consiste à rechercher les fautes qu'il peut contenir. Pour cela, le testeur va écrire puis exécuter des cas de test qui produiront chacun un verdict. Le verdict d'un cas de test est son résultat sous forme booléenne, un cas de test *"passe"* ou il *"échoue"*. Un cas de test passe si le résultat obtenu après exécution du programme sous test est correct par rapport à la spécification du programme, sinon il échoue. La spécification d'un programme est l'ensemble des exigences que ce programme doit satisfaire. Un cas de test est principalement composé de trois éléments : une initialisation, qui conduit le programme sous test dans un état souhaité ; des données de test qui serviront d'entrée du programme sous test ; et un oracle. L'oracle est un programme chargé de produire le verdict. Un oracle est composé d'une donnée d'oracle et d'une fonction d'oracle. La fonction d'oracle analyse le résultat obtenu par l'exécution du programme sous test et utilise la donnée d'oracle pour produire le verdict.

L'Ingénierie Dirigée par les Modèles (IDM) est une technique qui place les modèles au cœur des différentes activités du développement logiciel. Un modèle est une abstraction d'un système, il peut être utilisé pour décrire sa structure ou son comportement par exemple. Cette abstraction permet notamment aux développeurs de ne pas prendre en compte les détails techniques de la mise en œuvre. Ils peuvent ainsi se concentrer sur les aspects fonctionnels. La réutilisation de modèles existants est également facilitée par cette abstraction. Un modèle est écrit dans un langage particulier, ce langage est décrit par un méta-modèle. Un modèle est conforme à un méta-modèle. Une machine à états hiérarchique permet par exemple de modéliser le comportement d'un système. Elle décrit

les différents états dans lesquels peut se trouver le système et les transitions qui entrainent un changement d'état.

Les modèles ne sont pas figés, ils évoluent au fil de leur utilisation, ils peuvent par exemple être combinés pour en obtenir de nouveaux. Le développeur peut également simplifier un modèle en enlevant certains détails. C'est le rôle des transformations de modèles, qui sont automatisées. Cette automatisation est un élément essentiel de l'IDM, pour pouvoir réutiliser ces transformations un grand nombre de fois. Une transformation de modèles peut, par exemple, réaliser la mise à plat de la machine à états hiérarchique. Le modèle d'entrée de la transformation est une machine à états hiérarchique, le modèle de sortie est une nouvelle machine à états décrivant le même comportement sans utiliser d'états composites.

Avant de pouvoir la réutiliser de multiples fois, une transformation de modèles doit être validée. Une solution pour valider une transformation est de la tester. Dans le contexte de l'IDM, la donnée de test ou modèle de test, est un modèle d'entrée de la transformation sous test. Le modèle obtenu est produit par l'exécution de la transformation sous test sur le modèle de test. L'oracle contrôle la correction de ce modèle obtenu et fournit le verdict du cas de test. Si le modèle obtenu est conforme à la spécification de la transformation, le test passe, sinon il échoue.

L'IDM est une technique de plus en plus répandue. Le besoin de disposer de transformations de modèles validées est grandissant, le test de transformations de modèles est le sujet de nombreuses études. Ces études portent sur la génération et la sélection de modèles de test, l'oracle, ou encore la mise au point de *frameworks* dédiés au test de transformations de modèles.

Dans cette thèse nous nous intéressons à l'étude de l'oracle. Contrairement à la génération et la sélection de modèles de test [Fleurey et al., 2009], peu d'études ont été consacrées à l'oracle. Il a pourtant été reconnu que l'oracle était une composante importante du test, et qu'il ne doit plus être négligé dans les études à venir [Bertolino, 2007, Staats et al., 2011].

Actuellement, il n'est pas possible de générer de manière automatique (sauf si le testeur dispose d'une spécification formelle de la transformation sous test) un oracle pour un cas de test. Nous pouvons envisager d'assister le testeur dans cette tâche essentiellement manuelle. Cette assistance peut être passive, et prendre la forme de fonctions d'oracles adaptées à ses besoins. Nous pouvons aussi envisager une assistance plus active en lui fournissant des indicateurs de la qualité de ses oracles ainsi que des pistes pour les améliorer. Ces pistes sont exploitées dans cette thèse.

Il existe des fonctions d'oracle spécifiques aux transformations de modèles [Mottu et al., 2008]. Plusieurs de ces fonctions d'oracles utilisent la comparaison de modèles. Une autre utilise des contrats, ces contraintes exprimées sur le modèle d'entrée, sur le modèle obtenu ou entre les modèles d'entrée et de sortie. A partir de là, nous nous posons la question **A** :

– **Les fonctions d'oracles existantes sont-elles adaptées à tous les besoins du testeur ?**

Le testeur peut choisir de ne contrôler qu'une partie du modèle obtenu ; parfois, ce choix lui est imposé. C'est par exemple le cas s'il n'est pas capable de prévoir pour un coût raisonnable la totalité du modèle attendu mais seulement une partie, c'est-à-dire *un modèle attendu partiel*. Ce modèle attendu partiel peut être produit par une précédente version de la transformation sous test. Dans un autre cas de figure, la transformation sous test a des sorties polymorphes ; il existe plusieurs modèles de sortie valides pour un modèle d'entrée donné. Ici, le modèle attendu partiel serait la partie commune à toutes les variantes du modèle de sortie. La mise à plat d'une machine à états hiérarchique est un exemple d'une telle transformation de modèles. Suivant le modèle d'entrée fourni, il peut exister plusieurs variantes du modèle de sortie, avec un nombre d'états finaux différent. Dans ce cas le testeur peut vouloir contrôler ce modèle de sortie obtenu sans tenir compte des états finaux. Dans un dernier cas de figure, si la transformation sous test effectue un ré-usinage, alors une partie du modèle d'entrée sera à l'identique dans le modèle de sortie. Dans ce cas, le testeur peut utiliser le modèle d'entrée comme modèle attendu pour contrôler la partie du modèle obtenu non modifiée par la transformation.

La fonction d'oracle la plus simple compare un modèle attendu au modèle obtenu, pour le contrôler entièrement. Le testeur doit donc obtenir d'une manière ou d'une autre, un modèle attendu entier. Si le modèle utilisé comme modèle attendu ne contient la valeur attendue que pour une partie seulement des éléments du modèle de sortie il s'agit d'un modèle attendu partiel. Si le modèle obtenu est comparé à un modèle attendu partiel, des différences seront observées à propos de la partie non présente dans le modèle attendu partiel et le test échouera. Pour utiliser cette fonction d'oracle avec une transformation aux sorties polymorphes, le testeur devrait utiliser pour chaque cas de test autant de modèles obtenus qu'il existe de variantes valides du modèle de sortie. Cela serait trop coûteux. Les fonctions d'oracles utilisant des contrats ne sont pas non plus appropriées pour obtenir un tel verdict partiel. Écrire ces contrats est aussi complexe que de développer la transformation, et le risque d'erreur est aussi élevé que pour la transformation sous test.

Les fonctions d'oracles existantes n'étant pas adaptées pour contrôler uniquement une partie d'un modèle obtenu, nous en proposons une nouvelle. Nous utilisons une *comparaison de modèles filtrée*. Nous comparons les modèles puis nous filtrons le résultat de cette comparaison pour retirer une partie des différences observées. Le testeur compare le modèle obtenu avec un modèle attendu, partiel ou non. Il filtre ensuite le résultat de cette comparaison pour retirer toutes les différences observées à propos de la partie du modèle obtenu qui ne l'intéresse pas. Le test passe si aucune différence n'est présente dans le résultat filtré de cette comparaison, sinon il échoue.

Avec cette nouvelle fonction d'oracle, nous offrons au testeur un nouvel outil pour créer des oracles. Cet outil vient compléter les fonctions d'oracles déjà existantes. Pour aller plus loin, nous nous intéressons à la qualité des oracles créés avec ces fonctions

d'oracles. A la différence des techniques de vérifications formelles, un testeur ne peut être certain que la transformation sous test ne contient plus aucune faute. Si tous ses tests passent, ils n'en ont détecté aucune. C'est pourquoi il est important d'écrire des cas de test, et donc des oracles de qualité. Nous nous posons ainsi la question **B** :

– **Comment évaluer la qualité d'un oracle pour le test de transformations de modèles ?**

Dans le cadre du test de programmes, la couverture de code et l'analyse de mutation sont deux des techniques utilisées pour qualifier des cas de test. La mesure de la couverture de code part du principe qu'un ensemble de cas de test de qualité fait intervenir la totalité du code du programme sous test. Ainsi, un outil dédié va mesurer quelles parties du programme sous test interviennent lors de l'exécution d'un cas de test. Le but étant qu'après exécution de tous les cas de test, il n'y ait pas de partie du programme qui ne soit pas intervenue. Mesurer la couverture de la spécification consiste à s'assurer que les cas de test font intervenir l'ensemble de la spécification du programme sous test. L'analyse de mutation, quant à elle, consiste à injecter volontairement des fautes dans le programme sous test et vérifier que les données de test détectent ces fautes.

Dans le contexte du test de transformations de modèles, mesurer la couverture de la spécification revient généralement à mesurer la couverture du méta-modèle d'entrée de la transformation par les données de test [Fleurey et al., 2009, Sen et al., 2012]. Cependant la qualification de l'oracle a fait l'objet de moins d'études. L'analyse de mutation est aussi utilisée pour qualifier les oracles d'une suite de tests [Jézéquel et al., 2001, Mottu et al., 2006b]. Le testeur crée des mutants, des versions erronées de la transformation sous test. Il injecte une unique faute dans chaque mutant. Les mutants transforment ensuite les modèles de test en modèles obtenus potentiellement erronés. Un mutant est tué, si au moins un de ses modèles obtenus est différent de celui produit par la transformation sous test. Pour évaluer les oracles, le testeur ne conserve que les mutants tués par les données de test. Les oracles contrôlent ensuite les modèles produits par les mutants. Si un test échoue, le mutant est tué. Plus le nombre de mutants tués par les oracles est élevé et meilleure est la qualité des oracles.

L'utilisation de l'analyse de mutation pour évaluer la qualité d'oracles de test de transformations de modèles a quatre inconvénients majeurs. Le premier inconvénient est que les mutants sont créés à la main par le testeur. Des opérateurs de mutation décrivent les fautes susceptibles d'être présentes dans une transformation de modèles. Le testeur crée chaque mutant en appliquant ces opérateurs. Le deuxième inconvénient de l'analyse de mutation est qu'elle est dépendante du langage de transformation utilisé. Les opérateurs de mutation sont indépendants, mais leur mise en œuvre est différente selon le langage de transformation. Le troisième inconvénient majeur de l'analyse de mutation est son coût. Tout d'abord à cause de la création manuelle des mutants par le testeur. Ensuite, à cause du temps d'exécution des mutants sur tous les modèles de test. Enfin, il faut prendre en compte le temps nécessaire pour effectuer les comparaisons entre les modèles de sortie produits par les mutants avec ceux de la transformation sous test. Le quatrième inconvé-

nient concerne l'exploitation par le testeur du résultat de cette évaluation pour améliorer la qualité de ses oracles. La mesure fournie par l'analyse de mutation est le rapport entre le nombre de mutants tués par les oracles sur le nombre de tués par les modèles de test lors de la phase initiale. Ce score est un indicateur de la qualité des oracles. Le testeur peut difficilement analyser les mutants vivants pour améliorer ses oracles.

Staats et al. [Staats et al., 2011] proposent de mesurer la puissance d'un oracle. Ils définissent la puissance d'un oracle comme sa capacité à détecter des fautes ; plus un oracle est puissant et plus il est capable de détecter des fautes. Nous proposons d'évaluer la qualité l'ensemble des oracles d'une suite de tests en mesurant sa puissance. Le score du mutation est lui aussi une évaluation de cette capacité à détecter des fautes ; plus le score de mutation est élevé et plus l'oracle détecte de fautes.

Nous considérons que la qualité d'un ensemble d'oracles est liée à sa couverture du méta-modèle de sortie. Notre proposition est que mieux un ensemble d'oracles couvre le méta-modèle de sortie et plus il est capable de détecter de fautes, et est donc plus puissant. Parmi plusieurs ensembles d'oracles, celui qui a la puissance la plus grande est de meilleure qualité. La mesure de la couverture du méta-modèle de sortie par un ensemble d'oracles fournit deux résultats : un taux de couverture et la liste des éléments du méta-modèle qui ne sont pas couverts par les oracles. Le taux de couverture est un indicateur de la qualité des oracles. Pour tenter d'améliorer la couverture, et donc les oracles, nous proposons au testeur d'essayer de couvrir les éléments de cette liste.

Les contributions de cette thèse s'articulent autour des deux questions posées. Tout d'abord, pour répondre à la question **A**, nous proposons d'aider le testeur en lui fournissant une nouvelle fonction d'oracle complémentaire de celles déjà existantes. Cette nouvelle fonction d'oracle lui permet de ne contrôler qu'une partie d'un modèle obtenu. Une telle fonction d'oracle est utile, par exemple, dans les cas où prévoir la valeur de la totalité du modèle de sortie n'est pas possible pour un coût raisonnable. Nous avons développé un outil permettant de mettre en œuvre cette nouvelle fonction d'oracle. Le testeur définit à l'aide de *patterns* la partie du modèle obtenu qui ne l'intéresse pas, celle qu'il ne souhaite pas contrôler. Il compare ensuite le modèle obtenu à un modèle attendu partiel qu'il crée. Il peut aussi utiliser un modèle attendu non partiel s'il peut en obtenir un. Après cela il filtre le résultat de cette comparaison à l'aide des *patterns*. Le test passe si le résultat de la comparaison filtrée est vide. Pour valider cette contribution nous avons expérimenté deux cas d'étude, nous montrons qu'il est possible avec notre approche de ne contrôler qu'une partie du modèle obtenu. Nous avons créé 94 modèles de test et avons obtenus 94 verdicts partiels qui nous ont permis de détecter 4 bugs dans deux transformations. Nous avons défini 94 modèles attendus partiels composés de 2 632 éléments, soit 70 % de moins que pour l'approche classique de comparaison avec des modèles attendus entiers. Cette contribution a fait l'objet de plusieurs publications, à ICMT 2013 [Finot et al., 2013a], à CIEL 2012 [Finot et al., 2012a], au PhD Workshop de ICTSS 2012 [Finot et al., 2012c], ainsi qu'aux journées du GDR GPL 2012 [Finot et al., 2012b].

Pour fournir une assistance active au testeur, et pour répondre à la question **B**, nous nous intéressons à l'évaluation de la qualité de l'ensemble des oracles d'une suite de tests. Nous considérons la qualité d'un ensemble d'oracles comme sa capacité à détecter des fautes dans la transformation sous test. Nous proposons une métrique pour évaluer cette capacité à détecter des fautes. Nous mesurons la couverture de la spécification de la transformation par les oracles. Nous proposons également d'aider le testeur à améliorer ses oracles en lui proposant des pistes à explorer. Nous avons développé un outil permettant de mesurer la couverture d'un méta-modèle par un ensemble d'oracles, et donc de fournir un indicateur de la qualité de cet ensemble d'oracles. Nous avons également validé notre approche par des expériences sur deux transformations de modèles. Nous avons défini des cas de test pour ces deux transformations et nous avons évalué la qualité des ensembles d'oracles à l'aide de notre outil. Pour chaque transformation nous avons défini plusieurs ensembles d'oracles. Nous avons ensuite utilisé l'analyse de mutation pour qualifier à nouveau ces mêmes oracles. Nous avons créé des mutants de nos transformations sous test et utilisé nos oracles pour contrôler les modèles de sortie produits par les mutants. Les ensembles d'oracles ayant une meilleure couverture du méta-modèle de sortie ont également un score de mutation plus élevé. Cette contribution a été présentée à AMT 2013 [Finot et al., 2013b].

La suite de ce mémoire est organisée de la façon suivante : Dans le chapitre 2 nous présentons l'état de l'art du domaine. Le chapitre 3 est consacré à notre première contribution, la définition d'une nouvelle fonction d'oracle partielle pour le test de transformations de modèles. Nous présentons dans le chapitre 4 notre seconde contribution, une approche pour évaluer la qualité d'un oracle pour le test de transformations de modèles. Enfin dans le chapitre 5 nous concluons et discutons des perspectives de ce travail.

1.2 Cas d'étude

Tout au long de cette thèse, pour valider nos contributions, nous menons des expériences sur deux cas d'étude principaux. Ces deux transformations sont :

- la mise à plat d'une machine à états UML ;
- la traduction d'un diagramme d'états UML en un programme CSP modélisé.

1.2.1 Mise à plat d'une machine à états UML

Le premier cas d'étude sur lequel nous avons travaillé est une transformation qui réalise la mise à plat d'une machine à états. Après une présentation de cette transformation, nous décrivons les mises en œuvre avec lesquelles nous avons travaillé.

1.2.1.1 Présentation

Cette transformation nous sert d'exemple tout au long de cette thèse.

Une machine à états est un automate utilisé pour décrire le comportement d'un système. La machine à états fait partie des diagrammes de la norme UML. Elle se compose essentiellement d'états, qui représentent les états dans lesquels peut se trouver le système, et de transitions, qui indiquent un changement d'état. Une transition peut être déclenchée par un évènement, être conditionnée par une garde (une expression booléenne) et produire un effet.

Une machine à états hiérarchique, comme son nom l'indique peut être composée de plusieurs niveaux imbriqués les uns dans les autres. Cette hiérarchisation est représentée à l'aide d'états composites. Un état composite est un état du système qui, à la différence d'un état élémentaire peut contenir une nouvelle machine à états.

La mise à plat d'une machine à états hiérarchique, consiste à supprimer les états composites présents dans le modèle d'entrée. Plus précisément, le modèle de sortie décrit le même comportement que le modèle d'entrée sans utiliser d'états composites.

Les modèles de sortie de cette transformation sont polymorphes, pour un modèle d'entrée donné il existe plusieurs modèles de sortie valides. En effet d'après la norme UML [1], une machine à états peut avoir plus d'un état final. Ainsi le modèle de la figure 2.8 (page 28) est lui aussi un modèle de sortie valide pour la mise à plat de la machine à états de la figure 2.3 (page 25). Au lieu de n'avoir que les états B et C reliés par une transition à un même état final comme dans le modèle de la figure 2.7 (page 25), ici ils sont reliés à deux états finaux distincts. Le comportement décrit est le même dans les trois modèles.

1.2.1.2 Mise en œuvre

Pour nos expériences, nous utilisons deux mises en œuvre distinctes de cette transformation (une complète et une simplifiée).

Version complète La première mise en œuvre que nous utilisons réalise la mise à plat d'une machine à états conforme au méta-modèle UML complet. La figure 1.1 présente une partie du méta-modèle UML décrivant la structure des machines à états UML.

Cependant nous n'utilisons pas toute l'expressivité proposée par le méta-modèle UML. Nous ajoutons les préconditions suivantes sur les modèles d'entrée :

– ils ne contiennent pas d'états orthogonaux (des états composites qui contiennent plusieurs sous machines à états parallèles) ;

1. http://www.omg.org

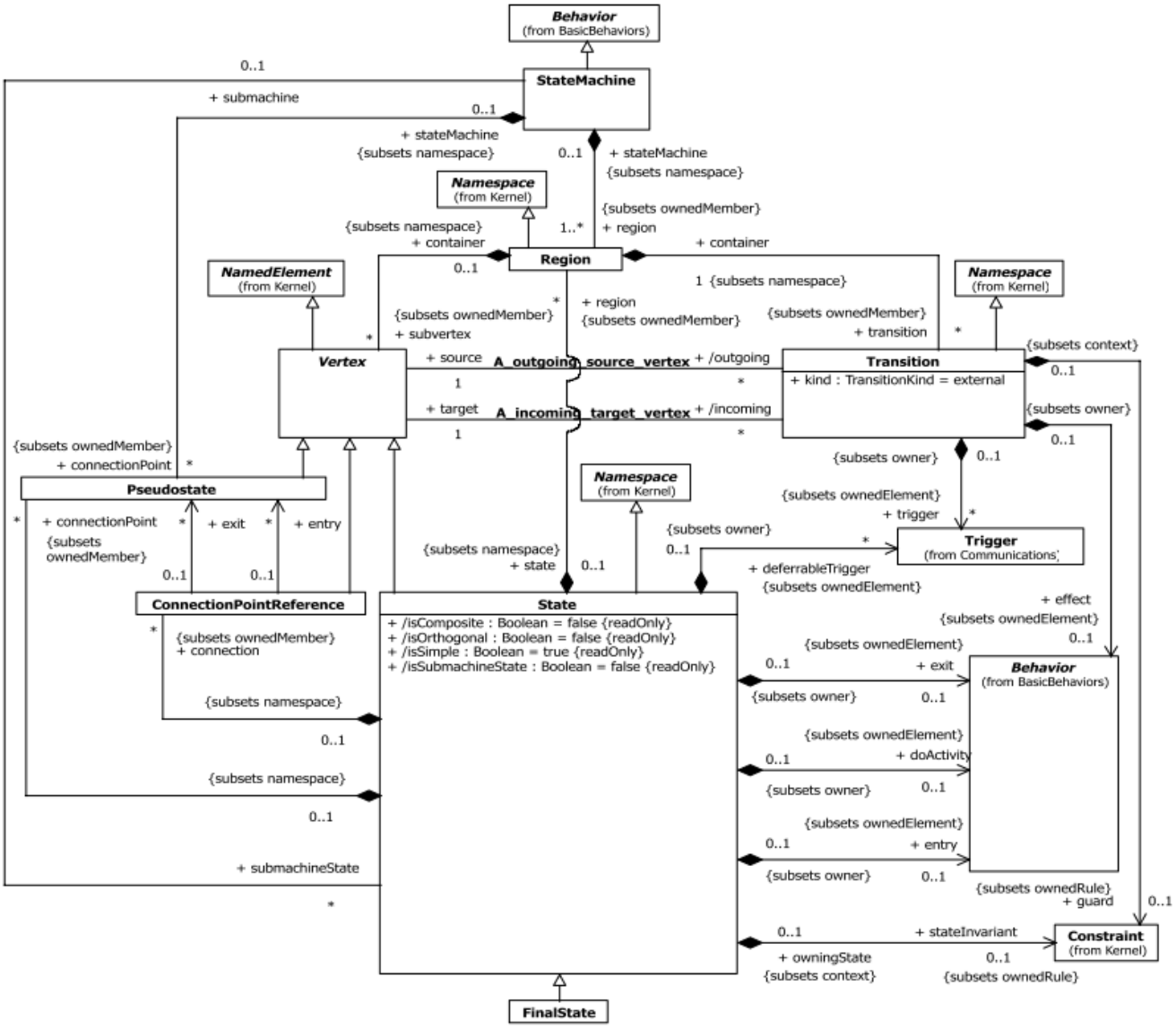


Figure 1.1 – Fragment du méta-modèle UML concernant les machines à états hiérarchiques (source OMG : http://www.omg.org/spec/UML/2.4.1/Superstructure/)

– ils ne contiennent pas de pseudo-états autres que les états initiaux ;
– ils ne contiennent pas d'hyper arc ;
– les transitions sortant d'un état composite contenant un état final n'ont pas de déclencheurs.

Cette mise en œuvre de la transformation a été réalisée en Kermeta par Holt et al. [Holt et al., 2009].

Version simplifiée Nous avons également mené des expériences avec une autre mise en œuvre de la transformation. Cette mise en œuvre utilise une version simplifiée du méta-modèle UML. Ce méta-modèle simplifié est présenté en figure 2.4 (page 25).

Les modèles manipulés par cette version de la transformation sont plus petits ; la mise en œuvre de cette transformation est aussi plus courte.

Nous avons utilisé cette mise en œuvre de la transformation pour valider une contribution à l'aide de l'analyse de mutation. Les mutants étant créés manuellement cela nous a paru plus réaliste d'utiliser cette mise en œuvre plus légère de la transformation.

Nous avons mis en œuvre cette version de la transformation en Kermeta [Finot et al., 2013b].

1.2.2 Diagramme d'activité UML vers CSP

Le diagramme d'activités de la norme UML, est utilisé pour représenter graphiquement le comportement d'une méthode. CSP est une algèbre de processus, proposée par Hoare [Hoare, 1978] et permettant de modéliser l'interaction entre systèmes. La transformation de modèles **T2** proposée par Bisztray et al. [Bisztray et al., 2007] transforme un diagramme d'activités en un modèle représentant un programme décrit en CSP. Les méta-modèles d'entrée et de sortie sont présentés respectivement par les figures 1.2 et 1.3.

Nous avons identifié deux règles de la transformation **T2** produisant des variations du modèle de sortie (qui en devient polymorphe) :

1. Un branchement conditionnel devient une condition n-aire. Les différents opérandes peuvent être permutés et différents branchements peuvent se combiner de différentes manières. Les auteurs précisent que d'après la norme UML, si les conditions des gardes sont disjointes, différentes organisations syntaxiques sont sémantiquement équivalentes.
2. Une barre de synchronisation devient une combinaison d'opérateurs de concurrence : l'opérateur de concurrence indique que plusieurs processus sont parallélisés. L'ordre dans lequel ces processus parallèles sont présentés ne change pas la sémantique du programme.

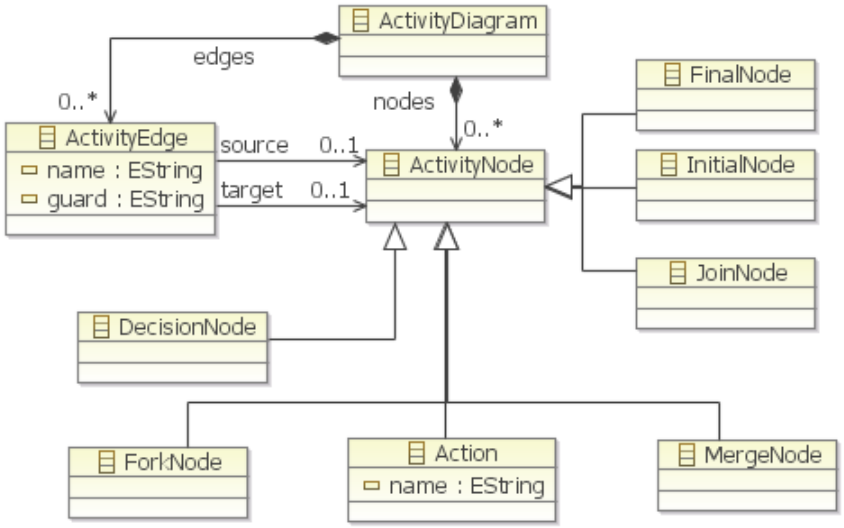


Figure 1.2 – Méta-modèle simplifié du diagramme d'activité UML

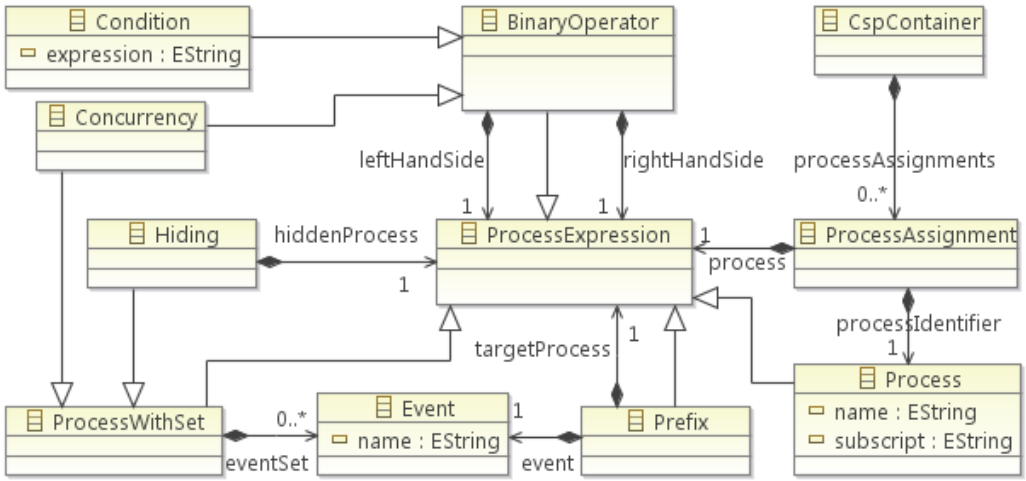


Figure 1.3 – Méta-modèle du langage CSP

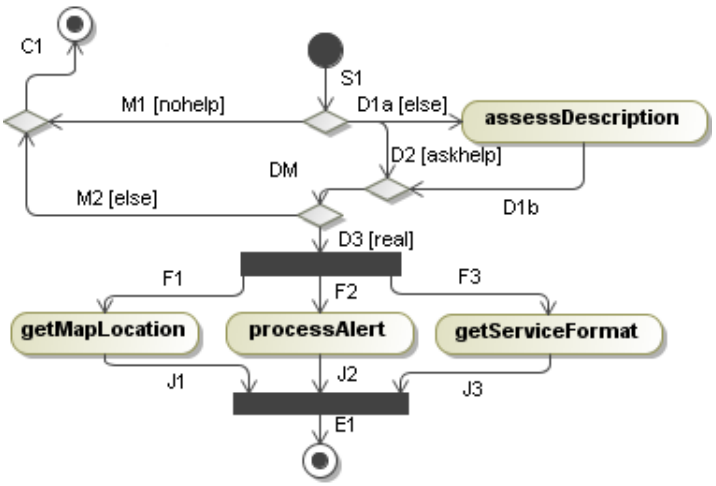


Figure 1.4 – Exemple de diagramme d'activités

```
S1=M1 ⧏nohelp⧐(D2 ⧏askhelp⧐ D1a)
M1 = C1
D2 = DM
D1a = assessDescription → D1b
D1b = DM
DM = D3 ⧏real⧐ M2
M2 = C1
C1 = SKIP
D3 = F1 ||| F2 ||| F3
F1 = getMapLocation → J1
F2 = processAlert → J2
F3 = getServiceFormat → J3
J1 = processJoin → E1
J2 = processJoin → SKIP
J3 = processJoin → SKIP
E1 = SKIP
```

Figure 1.5 – Programme CSP correspondant au diagramme d'activité de la figure 1.4

La figure 1.4 présente un exemple de diagramme d'activités possédant un branchement conditionnel avec trois choix différents, et une barre de synchronisation. La figure 1.5 présente une traduction possible de ce diagramme d'activité en code CSP. La transformation du branchement conditionnel recevant l'arc S4, peut produire deux résultats différents :

- `S4 = M1` $\not<$*nohelp*$\not>$[2] `(D2` $\not<$*askhelp*$\not>$ `D1a)` ou
- `S4 = D2` $\not<$*askhelp*$\not>$ `(M1` $\not<$*nohelp*$\not>$ `D1a)`

La transformation de la barre de synchronisation se transforme en trois processus parallèles `(F1, F2, F3)`, qui peuvent s'organiser de six manières différentes :

- `D3 = F1 ‖ F2 ‖ F3`
- `D3 = F1 ‖ F3 ‖ F2`
- `D3 = F2 ‖ F1 ‖ F3`
- `D3 = F2 ‖ F3 ‖ F1`
- `D3 = F3 ‖ F1 ‖ F2`
- `D3 = F3 ‖ F2 ‖ F1`

Dans cet exemple, deux éléments du modèle d'entrée introduisent des variations possibles en sortie. Pour définir les modèles de sortie possibles, il faut combiner ces deux possibilités de variations. La transformation du branchement conditionnel peut produire deux résultats, et celle de la barre de synchronisation en produit six, ce qui porte à douze le nombre de modèles de sortie valides. Pour définir l'oracle du test de cette transformation de modèles pour ce modèle d'entrée, le testeur devra définir les douze modèles attendus valides. L'oracle ainsi défini devra contrôler si le modèle obtenu fait bien partie de ces douze variantes, en le comparant avec chacune.

Ces cas d'étude montrent que le test d'une transformation de modèles peut nécessiter de définir plusieurs modèles attendus pour une unique donnée de test.

Nous avons développé cette transformation en ATL.

2. Ici *nohelp* est l'expression d'une condition ayant comme opérandes M1 et une autre condition

CHAPITRE 2

État de l'art

Dans ce chapitre nous présentons le contexte de cette thèse et dressons un état de l'art des travaux publiés à ce jour. Cet état de l'art est concentré sur deux axes principaux, le *test logiciel* et *L'Ingénierie Dirigée par les Modèles*. Dans la Section 2.1, nous présentons les principes généraux du test logiciel. Dans la Section 2.2, nous présentons l'Ingénierie Dirigée par les Modèles. Dans la Section 2.3, nous nous intéressons aux travaux existants sur le test de transformations de modèles. Dans la Section 2.4, nous synthétisons les problématiques mises en évidence dans ce chapitre.

2.1 Test logiciel

Un programme est écrit pour réaliser une fonctionnalité, tout en respectant les critères de qualité exigés. Avant d'utiliser et réutiliser un programme il est important de s'assurer que ce dernier est correct par rapport à sa spécification. Le test est une technique de validation logicielle. Dans cette section nous présentons tout d'abord les principes du test logiciel, avant de discuter plus précisément d'une de ses composantes : l'oracle. Nous discutons ensuite de l'ingénierie des besoins pour enfin nous intéresser à la qualité des tests.

2.1.1 Processus du test

Le test est une activité dont le but est de détecter les fautes présentes dans un programme. Un test peut être *statique* ou *dynamique*. Avec un test statique, le programme sous test est validé sans être exécuté. Cette validation se fait par analyse statique de code, automatiquement ou manuellement. Avec un test dynamique, le programme sous test est exécuté pour provoquer sa défaillance et révéler la présence d'une faute.

Les techniques de test utilisées sont liées à la manière dont est conçu et mis en œuvre le programme sous test. Par exemple, dans le cycle classique de développement logiciel en V (Figure 2.1) il existe plusieurs types de test correspondant aux différentes phases de développement. Un test unitaire vérifie une unité du programme sous test : une classe ou une méthode dans le cas de programmation orientée objet. Un test d'intégration, lui vérifie les interactions entre plusieurs unités du programme. Les tests de validation ou tests systèmes vérifient la totalité du système en intégrant les groupes d'unités validés par les tests d'intégration. Les tests de recettes quant à eux ont pour but de valider le système par rapport aux attentes du client, définies dans la spécification.

Pour tester un programme, le testeur écrit une *suite de test*, composée de *cas de test*. Un cas de test correspond à une exécution du programme à tester, le programme sous test ou *System Under Test (SUT)*. Pour chaque cas de test, comme le montre la Figure 2.2, le testeur sélectionne une *donnée de test* (DT) qui est une donnée d'entrée du programme sous test. Il exécute ensuite le programme sous test sur la donnée de test et produit un

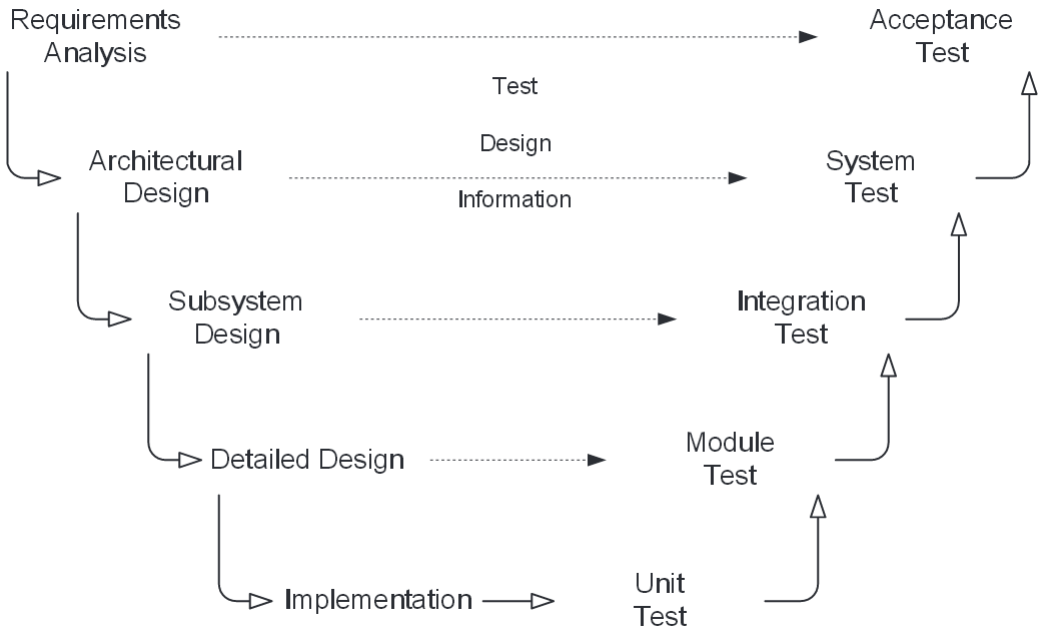


Figure 2.1 – Le cycle de développement en V (source [Ammann and Offutt, 2008])

résultat (Res). *L'oracle* du cas de test contrôle le résultat produit et fournit le *verdict* du cas de test. Nous reparlerons en détail de l'oracle dans la prochaine section. Si le résultat est correct par rapport à la spécification, le test *passe*, sinon il *échoue*. Lorsqu'une faute est détectée par l'oracle, il faut la corriger. Lorsque cette correction introduit une nouvelle faute, il y a *régression*.

Un test dynamique peut être *fonctionnel* ou *structurel* [Beizer, 1990] :

- Un test fonctionnel ne considère que la spécification du programme sous test. Le programme est considéré comme enfermé dans une "*boîte noire*".
- Un test structurel exploite lui la structure du programme sous test, qui est considéré comme étant dans une "*boîte blanche*".

Le test peut également être utilisé pour s'assurer que la correction d'une faute ou l'ajout d'une nouvelle fonctionnalité au système sous test ne perturbe pas les fonctionnalités existantes. Il s'agit de test de *non régression*.

2.1.2 Oracle du test

Nous détaillons tout d'abord les principales caractéristiques d'un oracle. Ensuite nous nous intéressons en particulier aux fonctions d'oracle. Enfin nous considérons les techniques existantes pour générer automatiquement l'oracle d'un cas de test.

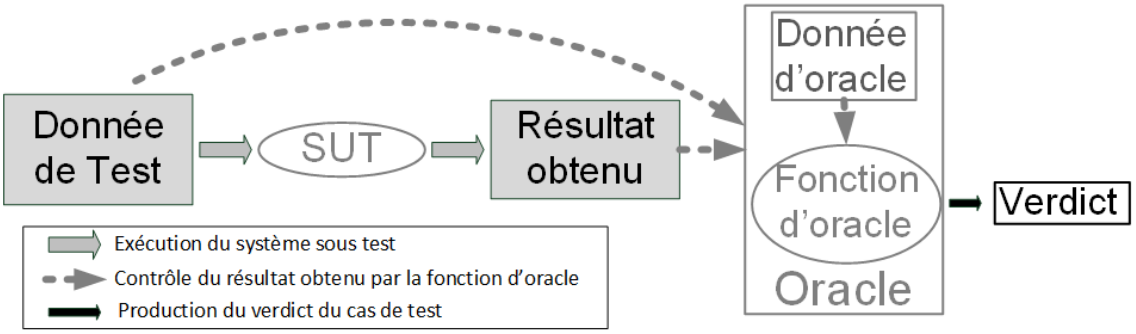


Figure 2.2 – Un cas de test

2.1.2.1 Caractéristiques d'un oracle

Staats et al. [Staats et al., 2011] constatent que peu de travaux de recherche ont une approche globale du test et se concentrent uniquement sur certains aspects. Selon eux, il serait intéressant de considérer les relations entre les différentes composantes du test, comme par exemple entre l'oracle et les données de test ou le SUT. Ils proposent de décrire un système de test comme un N-Uplet $(P, S, T, O, corr, corr_t)$ où :

- S est est un ensemble de spécifications ;
- P est un ensemble de programmes ;
- T est un ensemble de données de test ;
- O est un ensemble d'oracles ;
- $corr \subseteq P \times S$;
- $corr_t \subseteq T \times P \times S$.

O est un ensemble d'oracles (prédicats) o tel que $o : T \times P \rightarrow Boolean$. Pour $t \in T$ et $p \in P$, $o(t, p) = true$ signifie que le test passe. $corr(p, s) = true$ signifie que p est correct vis-à-vis de s. $corr_t(t, p, s) = true$ signifie que l'exécution de p sur la donnée de test t est conforme à la spécification s.

Staats et al. [Staats et al., 2011] citent deux caractéristiques d'un oracle : il peut être *complet*, et/ou *sain*. Un oracle est *complet* par rapport à p et s si pour chaque donnée de test t :

$$corr_t(t, p, s) \Rightarrow o(t, p)$$ [1]

Si le programme est correct par rapport à la spécification s sur la donnée de test t, alors le test passe. Si l'oracle est complet, il n'y a pas de faux positifs.

Un oracle est *sain* (*sound*) si :

$$o(t, p) \Rightarrow corr_t(t, p, s)$$

1. Pour un prédicat $Q(x)$, nous écrivons simplement $Q(x)$ au lieu de $Q(x) = vrai$

Si le test passe, alors le programme est correct par rapport à la spécification s sur la donnée de test t. Donc s'il y a une faute dans le programme, le test échoue.

Un oracle est *parfait*, s'il est à la fois complet et sain :

$$\forall t, o(t,p) \Leftrightarrow corr_t(t,p,s)$$

Douglas Hoffman [Hoffman, 1998] discute lui aussi de la complétude d'un oracle, mais il en donne une autre définition. Selon lui cette complétude peut aller d'une absence de prédiction du résultat attendu jusqu'à une duplication complète (une seconde mise en œuvre du SUT). Si l'oracle est une seconde mise en œuvre du système sous test, alors il n'y aura pas de faux positifs. De la même manière, s'il y a une faute dans le programme sous test, le test échouera. Un oracle complet selon la définition d'Hoffman est donc un oracle parfait selon les critères de Staats et al..

Staats et al. proposent également de comparer des oracles selon leur puissance. Selon eux, un oracle o_1 est plus puissant qu'un autre o_2 par rapport à un ensemble de test TS (écrit $o_1 \geq_{TS} o_2$) pour un programme p et sa spécification s si :

$$\forall t \in TS, o_1(t,p) \Rightarrow o_2(t,p)$$

Pour un cas de test donné, si o_1 ne détecte pas de faute, alors o_2 non plus. Donc si o_2 détecte une faute, alors o_1 aussi. Mais si o_2 ne détecte pas de faute, o_1 qui est plus puissant, peut lui en détecter une.

La création d'un oracle se fait en boîte noire ; le testeur ne doit pas être influencé par les éventuelles erreurs commises par le développeur.

2.1.2.2 Fonctions d'oracle

Comme le montre la Figure 2.2 un oracle se compose de deux éléments, une *donnée d'oracle* et une *fonction d'oracle*. La fonction d'oracle est chargée de produire le verdict du test, elle contrôle le résultat produit par le système sous test en utilisant la donnée d'oracle et, dans certains cas, la donnée de test. Il existe plusieurs types de fonctions d'oracle, nous les présentons ici et discutons leurs avantages et inconvénients lorsque nous les appliquons au test de transformations de modèles (Section 2.3.2, page 33).

Intuitivement, la fonction d'oracle la plus simple est une comparaison. L'oracle compare le résultat obtenu après exécution du système sous test au résultat attendu pour une exécution correcte.

Une autre possibilité, proposée par Meyer [Meyer, 1992] est l'utilisation de contrats. Un contrat est essentiellement une post-condition sur le système sous test. Un contrat exprime des contraintes sur le résultat produit ou sur les relations entre entrée et sortie. Dans ce dernier cas, en plus de la donnée d'oracle (ici les contrats écrits par le testeur) la fonction d'oracle a également besoin de la donnée de test pour vérifier les contraintes.

Polikarpova et al. [Polikarpova et al., 2009] ont mené une étude comparative entre des contrats écrits manuellement et d'autres inférés automatiquement. Ils ont tout d'abord évalué la qualité des contrats inférés. Ils ont considéré comme étant inintéressants les contrats répondant aux critères suivants :

- Deux variables qui ne sont pas liées sont mises en relation.
- Deux constantes sont comparées.
- Un contrat qui est impliqué par un autre au même point du programme, de manière triviale (sans connaître le code source).
- Un contrat qui concerne un point du programme autre que celui où il a été inféré.

Baresi et al. [Baresi and Young, 2001] mènent une étude sur les oracles de test. Ils identifient plusieurs types de langages permettant d'écrire des oracles, dont :

- Les mécanismes d'assertions embarqués dans des langages (comme Eiffel par exemple qui intègre des contrats).
- Les langages de spécification pure (par exemple Z).
- Les langages de vérification de traces.

Selon Elaine Weyuker [Weyuker, 1982], écrire l'oracle d'un cas de test n'est pas toujours possible. Dans cette situation, elle propose d'utiliser un oracle partiel qui va contrôler uniquement une partie de la sortie produite.

Une autre technique, la programmation de N-versions [Chen and Avizienis, 1995], n'a pas besoin d'oracles pour produire un verdict. Manolache et al. [Manolache and Kourie, 2001] proposent ainsi d'utiliser plusieurs mises en œuvre d'une même spécification, réalisées indépendamment les unes des autres. Après exécution de chacune des versions sur les données de test, les résultats produits sont comparés les uns aux autres. L'observation d'une différence entre les résultats produits par des mises en œuvre différentes révèle une faute dans au moins une des mises en œuvre. Le testeur recherche alors l'origine du problème, corrige la (les) mise(s) en œuvre erronée(s) et recommence. Le principal inconvénient de cette approche est la nécessité d'avoir plusieurs mises en œuvre de la transformation sous test. Le testeur peut ne pas avoir à sa disposition plusieurs versions de la transformation, et écrire lui-même ces autres mises en œuvre serait au moins aussi coûteux que d'écrire le programme sous test.

Une autre approche pour définir un cas de test sans utiliser d'oracle est le *test métamorphique*, étudié par Murphy et al [Murphy et al., 2009]. Cette approche suppose que le programme sous test a des propriétés métamorphiques : soient p le programme sous test, et x une de ses entrées. Le programme p a des propriétés métamorphiques s'il existe une fonction f telle qu'il est possible d'obtenir la valeur de $p(f(x))$ à partir de celle de $p(x)$. Dans ce cas, le testeur va calculer les valeurs de $p(x)$ et $p(f(x))$ à l'aide du programme sous test, et comparer la valeur obtenue pour $p(f(x))$ à celle calculée à partir de $p(x)$, si les deux valeurs sont différentes, il y a une faute dans le programme. La fonction sinus (sin) est un exemple de programme aux propriétés métamorphiques, en effet $sin(x) = sin(\pi - x)$. Si pour une valeur donnée de x, $sin(x) \neq sin(\pi - x)$, alors il y a une faute dans le programme. Pour appliquer cette approche le programme sous test doit donc avoir des propriétés métamorphiques ; or cela n'est pas immédiat comme propriété

de programme. Cette approche permet uniquement de détecter des erreurs, le testeur ne peut pas être certain que les valeurs obtenues soient valides.

2.1.2.3 Model-Based Testing et génération automatique de l'oracle

La définition et l'application d'une suite de tests est une tâche coûteuse, c'est pourquoi il est intéressant de pouvoir automatiser tout ou partie de ce processus. L'une des solutions proposée est de générer les cas de test à partir d'une modélisation du système sous test, c'est le *Model-Based Testing* [Dalal et al., 1999].

La première étape de ce processus est la construction d'un modèle du système par le testeur. Ce modèle, généralement construit manuellement par le testeur à partir de la spécification, est une abstraction du système sous test. Après validation du modèle, les cas de test sont ensuite extraits de ce dernier selon un critère choisi par le testeur.

Utting et al. [Utting et al., 2012] ont réalisé une étude des différentes approches et outils existants pour le Model-Based Testing. Ils classent notamment les approches étudiées selon la portée du modèle. Selon eux si le modèle spécifie les entrées possibles du système sous test, il peut aussi représenter le comportement attendu du programme sous test. Ainsi les cas de tests générés à partir de modèles ne spécifiant que les entrées ne contiennent que des données de test. Dans ce cas le testeur pourra utiliser des oracles basiques, le test passe si aucune exception n'est levée lors de l'exécution. S'il souhaite plus de précision, il lui faudra manuellement définir ses oracles, comme dans l'approche proposée par Dalal et al. [Dalal et al., 1999].

Dans les approches où le comportement du système est modélisé, il est possible d'extraire le résultat attendu pour l'exécution du système sur la donnée de test. Il est donc possible, dans ce cas, de générer automatiquement les oracles. C'est le cas de l'approche présentée par Jan Tretmans [Tretmans, 2008]. Il propose d'utiliser des systèmes d'états transitions pour modéliser le comportement du système. Pretschner et al. [Pretschner et al., 2005] proposent également de modéliser le comportement du système ; le modèle est transformé en un problème de satisfaction de contraintes pour la génération des tests.

Ces approches permettent de générer automatiquement des cas de test (données et oracles) à partir d'un modèle représentant le comportement du système. Cependant, le modèle est généralement construit par le testeur à partir des spécifications du système sous test, comme pour l'oracle. Ces approches ne résolvent donc pas le problème de la création d'un oracle ; elles le déplacent. Pour générer automatiquement des oracles, le testeur devra d'abord construire le modèle de comportement.

2.1.3 Ingénierie des besoins

Tester un système, c'est chercher à s'assurer qu'un logiciel est conforme à une spécification. Une spécification peut être vue comme un ensemble de besoins auxquels le système doit répondre. L'étude de cet aspect de la conception et du développement logiciel est l'objet de l'ingénierie des besoins.

D'après Attarha et Modiri [Attarha and Modiri, 2011] l'ingénierie des besoins inclut 5 activités : l'extraction des besoins (i), la discussion et l'analyse des besoins (ii), la documentation des besoins (iii), la validation des besoins (iv), et enfin la gestion des besoins (v).

L'analyse de scénarios est une composante de l'ingénierie des besoins. Hsia et al. [Hsia et al., 1994] décrivent l'analyse des scénarios comme le processus de compréhension, d'analyse et de description du comportement d'un système sous la forme d'utilisations particulières attendues pour le système. Le produit de ce processus est un document contenant un ensemble de scénarios corrects et validés. Ce document peut ensuite servir de guide pour la suite de la conception. Il peut également être utilisé pour la validation du système, de manière similaire au model-based testing.

Pour Regnell et al. [Regnell et al., 1995], l'analyse des besoins conduit à la génération d'un modèle à partir des besoins. Encore une fois ce modèle peut servir de base à la suite du développement. Ils évoquent par exemple la vérification et validation du système.

Toujours dans une optique de combinaison de l'ingénierie des besoins et du test, Rosenberg et al. [Rosenberg et al., 1998] proposent une série de métriques liées aux besoins identifiés et exprimés. Selon eux, des plans de tests sont écrits par le testeur à partir des besoins. Plusieurs des métriques proposées concernent le lien entre besoins et tests ; par exemple chaque besoin doit être testé au moins une fois.

2.1.4 Qualité des tests

Le fait que tous les tests passent est une condition nécessaire mais non suffisante pour conclure que le programme sous test est correct. Cela signifie que les cas de test n'ont détecté aucune faute. D'après Dijkstra [Dijkstra, 1972] *"Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence"* . Le test de programme est efficace pour démontrer la présence de fautes, mais il n'est pas adapté pour démontrer leur absence. C'est pourquoi, il est important de créer des cas de test de qualité qui sont représentatifs du programme sous test. Dans cette section, nous présentons des travaux qui concernent l'évaluation de la qualité de cas de test.

2.1.4.1 Qualification de données de test

Antonia Bertolino [Bertolino, 2007] dresse un bilan des avancées réalisées dans le test logiciel et dresse une liste de défis à résoudre. L'un de ces défis concerne l'efficacité des tests. Cet axe de recherche consiste à mettre en évidence l'efficacité des critères de sélection des données de test. Les données de test doivent mettre en évidence des fautes dans le programme sous test.

Pour être certain que le système sous test est correct, le testeur devrait écrire autant de cas de test qu'il existe d'entrées valides, ce qui n'est pas possible. Si l'un des paramètres du système est un entier, alors le nombre de valeurs possibles est déjà trop élevé, même s'il est limité par la manière dont sont encodés les entiers en mémoire. C'est pourquoi il est nécessaire de restreindre le nombre de cas de test, tout en s'assurant qu'ils soient représentatifs du système sous test.

Dans le cas général, trois types de techniques sont principalement utilisées : l'analyse partitionnelle, la mesure de couverture, et l'analyse de mutation.

L'analyse partionnelle [Ostrand and Balcer, 1988] est une technique en boîte noire pour la sélection de données de test. Le but de cette technique est d'identifier les ensembles de valeurs de chaque paramètre pour lesquelles le comportement du système sera le même. Pour chaque paramètre le testeur partitionne l'ensemble de ces valeurs possibles à l'aide de la spécification. Pour une partition donnée, le comportement du système sous test ne doit pas changer suivant la valeur choisie. Le testeur réalise ensuite le produit cartésien des ensembles de partitions pour chaque paramètre. Chaque n-uplet obtenu correspond au minimum à un cas de test, le testeur choisit une ou plusieurs valeurs. Le choix de ces valeurs se fait généralement aux limites de chaque partition. Ce choix part du constat que les fautes présentes dans un programme sont souvent dues au fait que le développeur n'a pas prévu le comportement du programme pour des valeurs aux limites. Par exemple, si le développeur écrit `a` $>$ `1` au lieu de `a` $\geq$ `1`, alors un seul cas de test avec la valeur 1 va permettre de détecter la faute.

Les techniques basées sur une mesure de couverture sont en boîte blanche. La plus simple consiste à mesurer le nombre de lignes de code du système sous test utilisées lors de l'exécution des cas de test. Cette technique permet de qualifier un ensemble de données de test existant, mais ne fournit pas de méthode pour les définir. D'autres approches utilisent la structure du programme, comme l'analyse de flot de contrôle. Dans ce cas, le testeur établit le graphe du flot de contrôle du système sous test. Il s'agit d'un graphe représentant les chemins qui peuvent être suivi par le système sous test lors de son exécution. Les arcs de ce graphe représentent les instructions de contrôle et les nœuds eux représentent des blocs d'instructions sans saut. La sélection des données de test se fait en couvrant ce graphe de flot de contrôle selon un critère choisi par le testeur. Le critère peut par exemple être de couvrir tous les nœuds du graphe ou tous les arcs, voire tous les chemins. Le testeur liste alors tous les chemins qui dans le graphe de flot de contrôle satisfont le critère sélectionné.

Enfin, l'analyse de mutation [DeMillo et al., 1978] est une autre technique de test en boîte blanche pour qualifier un ensemble de données de test existant. Le principe est ici d'introduire volontairement des fautes dans le programme sous test et de s'assurer que les données de test permettent de détecter ces fautes. Nous détaillons la mise en œuvre de l'analyse de mutation pour la qualification de données de test pour le test de transformations de modèles dans la section 2.3.3.1.

2.1.4.2 Qualification d'oracles

À la différence des données de test, le sujet de la qualification d'oracles a peu été étudié. Un autre des défis mentionnés par Antonia Bertolino [Bertolino, 2007] concerne l'oracle. Ce défi consiste à définir des méthodes efficaces pour obtenir et automatiser des oracles.

En l'état actuel des connaissances, comme nous l'avons mentionné dans la Section 2.1.2, la création d'oracles reste une tâche essentiellement manuelle. Il existe cependant des approches permettant de qualifier un ensemble d'oracles existant.

Tout comme pour la qualification de données de test, la qualification d'oracles se fait principalement par une mesure de couverture ou l'analyse de mutation.

Shuler et Zeller [Schuler and Zeller, 2011] proposent de qualifier des oracles pour du test unitaire à partir d'une mesure de couverture. Parmi les instructions qui ont une influence sur le résultat contrôlé par l'oracle, ils mesurent la proportion de celles réellement contrôlées par l'oracle. Ils proposent d'utiliser du *program slicing* pour identifier les instructions qui influencent le résultat contrôlé par l'oracle, puis parmi ces instructions ils mesurent la proportion réellement contrôlée par l'oracle. Selon eux, avec leur approche, le testeur ne chercherait pas à exécuter autant de code que possible mais à contrôler autant de résultats que possible. De cette manière il identifierait plus de fautes. Cette approche fournit également des indicateurs pour améliorer la qualité des oracles. Le testeur sait quelles instructions ne sont pas couvertes. Il doit donc améliorer ses oracles pour contrôler le résultat de ces instructions. La mise en œuvre de cette approche est spécifique au langage de programmation utilisé, comme pour toute approche utilisant la couverture de code. De plus, elle nous semble difficilement applicable dans le contexte qui nous intéresse, le test de transformations de modèles. Il est possible d'associer une instruction aux éléments du modèle de sortie sur lesquels elle a un impact, mais comment peut-on identifier quels éléments du modèle obtenu sont contrôlés par l'oracle ?

En plus d'être utilisée pour qualifier des données de test, l'analyse de mutation permet également de qualifier des oracles. C'est ce que proposent Jézéquel et al. [Jézéquel et al., 2001]. Le score de mutation obtenu reflète la capacité des oracles à détecter des erreurs dans les modèles de sortie produits par les mutants. De nouveau, nous détaillons ci-dessous l'utilisation de l'analyse de mutation pour qualifier des oracles pour

le test de transformations de modèle. Knauth et al. [Knauth et al., 2009] utilisent eux aussi l'analyse de mutation pour évaluer la qualité d'oracles à base de contrats.

Staats et al. [Staats et al., 2012] considèrent eux aussi que la création d'un oracle reste un processus essentiellement manuel, réalisé par le testeur. Plutôt que de vouloir générer l'oracle d'un test, ils proposent d'aider le testeur en réduisant le nombre de variables de sortie à contrôler. Ils utilisent pour cela l'analyse de mutation, ils considèrent uniquement les variables qui ont permis de tuer un mutant et les trient pour ne garder que les plus pertinentes.

2.1.4.3 Qualité globale des cas de test

D'autres approches cherchent à évaluer la qualité de cas de test de manière plus globale. Pretschner et al. [Pretschner et al., 2005] utilisent du model based testing. Ils comparent deux suites de test, l'une écrite manuellement par un testeur et l'autre générée automatiquement. Pour cette comparaison ils ont sélectionné deux critères : une mesure de couverture au niveau du modèle et de la mise en œuvre (i) et le nombre de fautes détectées (ii). Ainsi, selon eux la qualité d'une suite de test repose sur la représentativité des cas de test et la capacité des oracles à détecter des fautes.

Heimdahl et al. [Heimdahl et al., 2004] ont mené une étude similaire. Ils comparent des tests générés en utilisant un critère de couverture à d'autres générés de manière aléatoire et statistique. De nouveau, le critère retenu pour la comparaison est la capacités des cas de test à détecter des fautes dans le système sous test.

Nagappan et al. [Nagappan et al., 2005] utilisent des mesures statiques à la fois du code du programme sous test et des tests. Ces métriques sont réparties en deux catégories : quantification des tests (par exemple le nombre d'assertions par ligne de code) ainsi que complexité et orienté objet. A ces métriques ils ajoutent un code couleur (rouge, orange et vert) en fonction du résultat pour donner des indications au testeur.

Fraser et al. [Fraser and Zeller, 2010] proposent un outil permettant de générer automatiquement des suites de test unitaires pour des programmes Java. La génération des données de test repose sur un algorithme génétique. Pour l'oracle ils utilisent l'analyse de mutation. Lorsqu'un mutant est tué, ils créent une assertion correspondant à la différence observée. La valeur attendue utilisée pour cette assertion est celle produite par le programme sous test original. Le testeur devra par la suite contrôler chacune des assertions ainsi générées pour s'assurer qu'elle est correcte vis-à-vis de la spécification. Une erreur détectée dans une assertion indique un bug dans le programme sous test.

Reichhart et al. [Reichhart et al., 2007] présentent une approche et un outil pour évaluer la qualité de cas de tests unitaires. Cette approche repose sur la notion de *Test Smells*. Ces Test Smells ont été identifiés à la suite d'une étude empirique ; ils représentent du code de test difficile à maintenir, comme par exemple un test ou une méthode de test avec un nom qui ne veut rien dire. Même s'il est utile d'écrire les tests de manière à ce que

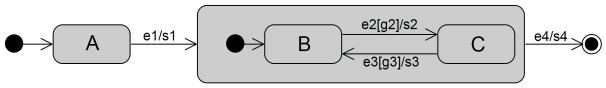


Figure 2.3 – Exemple de machine à états hiérarchique

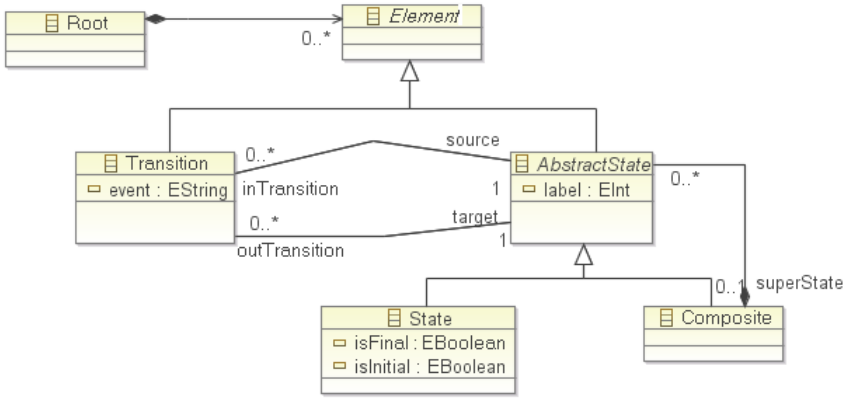


Figure 2.4 – Méta-modèle de la machine à états hiérarchique

leur maintenance soit aisée, cette méthode n'apporte aucune indication par rapport à la représentativité des cas de test ou la capacité des oracles à détecter des fautes.

2.2 Ingénierie Dirigée par les Modèles (IDM)

Dans cette section, nous discutons de l'IDM. Après avoir présenté ses principes, nous nous intéresserons plus particulièrement aux transformations de modèles. Nous présenterons ensuite plusieurs outils dédiés à l'IDM. Enfin nous détaillons les utilisations de la comparaison de modèles dans le contexte de l'IDM.

2.2.1 Principes de l'IDM

L'idée de l'IDM est de mettre les modèles au premier plan. Pour Bézivin [Bézivin, 2004], si pour le paradigme objet "tout est objet", pour l'IDM, "tout est modèle". Avec l'IDM, le développeur manipule principalement des modèles. Ils ne sont plus simplement utilisés dans les phases de conception comme UML.

La machine à états UML présentée dans la Figure 2.3, est un exemple de modèle qui représente un comportement. Un modèle est écrit dans un langage, ce langage est décrit par un méta-modèle. Un modèle est conforme à un méta-modèle qui définit sa structure. Le méta-modèle simplifié du modèle de la Figure 2.3 est présenté Figure 2.4.

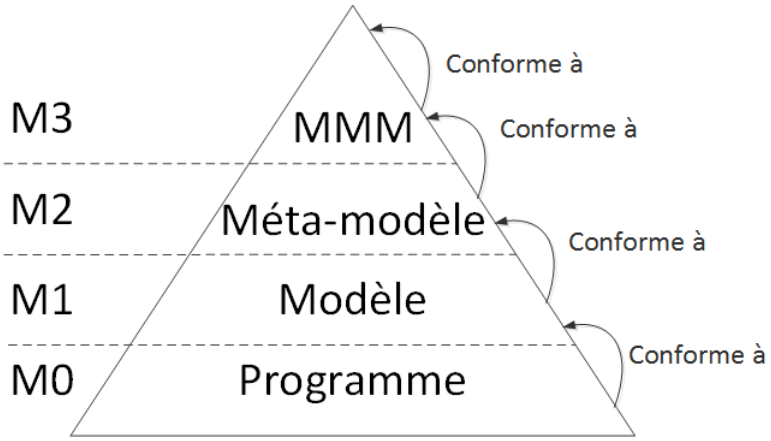


Figure 2.5 – Les différents niveaux de modélisation

L'initiative MDA (Model Driven Architecture) présentée par l'*Object Managment Group* (OMG) [Kleppe et al., 2003] est un aspect de l'IDM. Cette approche repose sur des standards tels que : Meta Object Facility (MOF), *XML Metadata Interchange* (XMI), *Object Constraint Language* (OCL).

Le MOF est un standard pour la représentation et la manipulation de méta-modèles. XMI est un standard basé sur XML, et créé par l'OMG pour l'échange d'informations et de méta-données UML. Il permet de sérialiser des modèles. OCL est un langage d'expression de contraintes conçu pour des modèles UML. Son utilisation a été généralisé aux modèles conformes au MOF.

D'après l'OMG il existe plusieurs niveaux de modélisation, comme présenté dans la Figure 2.5. Un modèle (niveau M1) représente un système (niveau M0) dont il est une abstraction. Un modèle est écrit dans un langage, ce langage est défini par un méta-modèle (niveau M2). Ce méta-modèle est lui-même écrit dans un langage, ce langage est lui aussi décrit par un méta-modèle, le méta-méta-modèle (niveau M3). Le méta-méta-modèle est instance de lui-même. Le MOF se situe au niveau M3.

Ainsi, la machine à états de la Figure 2.3 est un exemple de modèle du niveau M1 qui décrit le comportement d'un système. Ce modèle est instance du méta-modèle de la Figure 2.4 (niveau M2). Enfin ce méta-modèle est instance du méta-méta-modèle MOF.

2.2.2 Transformations de modèles

L'une des propositions du MDA est de définir les modèles indépendamment de la plateforme sur laquelle ils seront déployés, ce sont des *Platform Independant Model* (PIM) [Kent, 2002]. Après cela, le développeur fait évoluer le PIM pour l'adapter à sa plateforme de déploiement, il obtient ainsi un *Platform Specific Model* (PSM). Ce passage du PIM vers le PSM se fait à l'aide d'une transformation de modèles.

Figure 2.6 – Schéma de principe d'une transformation de modèles

Tom Mens [Mens, 2002] a étudié en détail les transformations de modèles. Une transformation est la génération automatique d'un modèle de sortie à partir d'un modèle d'entrée en fonction d'une définition de transformation. La définition d'une transformation est un ensemble de règles qui décrivent la manière dont un ou plusieurs éléments du modèle d'entrée sont transformés en un ou plusieurs éléments du modèle de sortie. Une transformation de modèles peut avoir un ou plusieurs modèles d'entrées ou de sortie. La Figure 2.6 schématise une transformation de modèles. Le modèle d'entrée M_{IN} est conforme au méta-modèle MM_{IN}. Le modèle M_{OUT} est le modèle obtenu pour la transformation de M_{IN}. Il est conforme au méta-modèle MM_{OUT}.

Selon Tom Mens, une transformation peut être endogène ou exogène, horizontale ou verticale. Si le modèle d'entrée et celui de sortie sont conformes à un même méta-modèle, la transformation est endogène. Au contraire, si les modèles d'entrée et de sortie sont conformes à des méta-modèles différents, elle est exogène. Avec une transformation horizontale, les modèles d'entrée et de sortie sont au même niveau d'abstraction. Si les modèles d'entrée et de sortie sont à des niveaux d'abstraction différents, alors la transformation est verticale.

La mise à plat d'une machine à états hiérarchique, telle que celle de la Figure 2.3, est un exemple de transformation de modèles. Avec cette transformation, le modèle de sortie est une machine à états décrivant le même comportement que le modèle d'entrée, sans aucun état composite. La Figure 2.7 est un modèle de sortie valide pour la mise à plat de la machine à états de la Figure 2.3.

D'autre part, dans certains cas, une transformation de modèles peut avoir des *sorties polymorphes*. C'est-à-dire que pour un modèle d'entrée il existe plusieurs variantes du modèle de sortie. Toutes ces variantes expriment la même sémantique avec des syntaxes

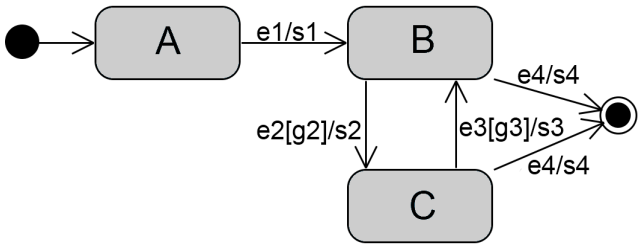


Figure 2.7 – Mise à plat possible de la machine à états de la figure 2.3

différentes. Ce polymorphisme des modèles de sortie vient de la spécification qui admet des variations sur certains éléments. Cependant, l'exécution de la transformation est déterministe et produit toujours le même modèle de sortie pour un modèle d'entrée donné. Soit te développeur fait un choix lors de la mise en œuvre soit cela est dû au comportement du langage de transformation utilisé et à son interpréteur déterministe. La mise à plat d'une machine à états est un exemple de transformation de modèles aux sorties polymorphes. En effet, dans la norme UML, rien n'oblige à n'avoir qu'un seul état final par machine. Ainsi, la machine de la Figure 2.8, avec ses deux états finaux distincts est également un résultat valide pour la mise à plat de la machine à états de la Figure 2.3, elle décrit le même comportement que la machine à états de la Figure 2.7.

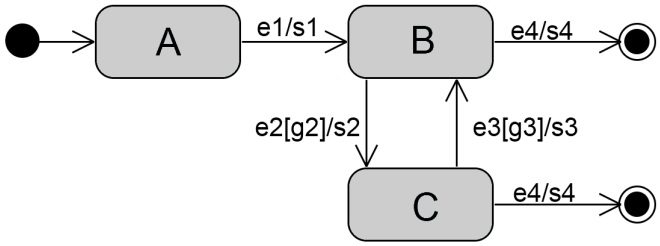


Figure 2.8 – Autre modèle de sortie valide pour la machine à états de la Figure 2.3

2.2.3 Outils dédiés à l'IDM

Eclipse propose un ensemble d'outils et de technologies pour manipuler les modèles [2]. Le cœur de ces technologies est l'*Eclipse Modeling Framework* (EMF), un *framework* permettant de créer des modèles. Ce *framework* fournit notamment *Ecore* un langage de méta-modélisation basé sur le MOF de l'OMG. Un méta-modèle est un modèle conforme à Ecore (instance d'Ecore). Les modèles sont enregistrés au format XMI.

En plus de ceux fournis par Eclipse, d'autres outils dédiés à l'IDM reposent sur EMF. *ATL* [Jouault et al., 2008] est un langage dédié à l'écriture de transformations de modèles. Ce langage repose sur la définition de règles de transformation par l'utilisateur. Kermeta [Muller et al., 2005] est un langage de manipulation de modèles. Il permet entre autres d'écrire des méta-modèles et des transformations de modèles. Ces deux langages manipulent des modèles au format XMI et dont les méta-modèles sont conformes au méta-modèle Ecore.

2. http ://eclipse.org/modeling/

2.2.4 Comparaison de Modèles et IDM

La comparaison de modèles est l'un des principaux outils de l'IDM. Elle est utilisée pour plusieurs fonctions d'oracle pour le test de transformations de modèles, mais pas uniquement.

2.2.4.1 Fonctionnement de la comparaison de modèles

Dans la comparaison de modèles trois phases sont distinguées, le calcul de la comparaison en elle-même, la manière dont le résultat de cette dernière sera représenté, et enfin la visualisation dudit résultat par l'utilisateur. Le calcul de la comparaison se décompose en deux phases distinctes. La première phase consiste à identifier les éléments communs aux deux modèles. Une fois ces points communs identifiés, la seconde phase s'attache à l'examen des différences.

Förtsch et al. [Förtsch and Westfechtel, 2007] proposent une liste de propriétés permettant d'évaluer un outil de comparaison de modèles, sa précision, son niveau d'abstraction, son domaine, son indépendance vis-à-vis des outils utilisés pour la création des modèles, son indépendance par rapport aux précédentes versions des modèles, son efficacité, l'ergonomie de la partie visualisation, ainsi que sa complexité.

2.2.4.2 Utilisations de la comparaison de modèles

La comparaison de modèles est utilisée pour répondre à différents défis auxquels est confronté l'IDM : (i) le développement collaboratif, (ii) l'évolution des modèles, (iii) leur composition, ou encore (iv) le test de transformations de modèles. Cette section présente la façon dont la comparaison de modèles est utilisée pour répondre aux problèmes posés par ces défis.

Développement collaboratif Une problématique du développement collaboratif est le travail de plusieurs collaborateurs sur un même élément. Si deux personnes modifient un même élément de manière différente, il faut faire un choix et sélectionner une version. Il faut donc identifier ce qui change entre ces versions. Si cet élément est un texte, un outil tel que *diff* peut calculer ces différences. Les modèles manipulés dans l'IDM sont plus durs à comparer que des fichiers texte. Ces modèles peuvent être représentés sous forme de texte, mais une comparaison textuelle n'est pas appropriée. D'après Förtsch et al. [Förtsch and Westfechtel, 2007] si le modèle est un fichier XML, la place d'un élément dans le fichier n'est pas déterminante. Donc une comparaison ligne par ligne ne serait pas forcément correcte. Mehra et al. [Mehra et al., 2005] ont mis au point des outils de comparaison dans une optique de développement collaboratif.

Composition de modèles La composition de modèles est proche du développement collaboratif. À partir de deux modèles différents, nous en créons un seul. D'après Kolovos et al. [Kolovos et al., 2006], identifier les éléments communs aux deux modèles sources est un prérequis pour que le résultat obtenu ne contienne aucun élément dupliqué. À la différence de l'évolution de modèles, la composition ne concerne pas forcément deux versions d'un même modèle. Une technique de comparaison à base d'identifiants n'est donc pas adaptée. Il est nécessaire d'utiliser une analyse syntaxique des différences, telle que celle proposée par Lin et al. [Lin et al., 2007] avec DSMDiff.

Évolution de modèles Lorsqu'un modèle évolue il est utile de connaître les différences entre deux versions distinctes, surtout pour l'évolution d'un méta-modèle. Des modèles préexistants ne sont plus forcément conformes à la nouvelle version du méta-modèle. Il faut analyser chaque mise à jour pour faire évoluer de manière similaire le modèle si besoin est [Cicchetti et al., 2008].

Certains outils de modélisation rajoutent des identifiants aux éléments des modèles créés. Dans un contexte d'évolution de modèles nous comparons deux versions d'un même modèle. Ces identifiants sont présents dans les deux modèles à comparer, nous pouvons utiliser ces identifiants comme base pour la comparaison. Cette méthode est plus efficace qu'une analyse syntaxique des modèles, mais elle est aussi dépendante de l'outil de modélisation utilisé. Cette méthode ne peut être utilisée que dans ce contexte d'évolution de modèles. Elle est inutilisable si la nouvelle version est créée indépendamment de la précédente. Même en utilisant le même outil les identifiants ne correspondraient pas et la comparaison serait faussée.

Génération de règles de transformations de modèles Pour écrire une transformation de modèles, le développeur doit établir un lien entre les méta-modèles d'entrée et de sortie. Lopes et al ; [Lopes et al., 2006] ont étudié la comparaison de méta-modèles. En particulier ils se sont intéressés à l'identification de leurs points communs. A partir de leurs travaux, ils ont conçu *SAMT4MDE* un outil qui permet de générer des règles de transformations ATL de manière semi-automatique [Lopes et al., 2005].

Test de transformations de modèles Comme nous le verrons plus loin, trois fonctions d'oracles parmi celles existantes pour le test de transformations de modèles (cf Section 2.3.2, page 33) ont recours à une comparaison de modèles. Plusieurs outils utilisent cette technique : (i) Lin et al. ont développé un moteur de test [Lin et al., 2005] basé sur DSMDiff, un moteur de comparaison de modèles, (ii) EUnit [García-Domínguez et al., 2011] compare des modèles à l'aide d'EMFCompare (conforme aux principes exposés par Cicchetti et al. [Cicchetti et al., 2007], le résultat de la comparaison est lui-même un modèle). De telles comparaisons de modèles requièrent des modèles attendus non partiels ; sinon la comparaison identifierait des différences

concernant la partie non contrôlée du modèle de sortie. Ces différences devraient être manuellement analysées pour obtenir un verdict. De plus, si la transformation a des sorties polymorphes, le modèle obtenu devra être comparé à chacune des variantes valides, il ne peut correspondre qu'à l'une d'entre elles. Ces fonctions d'oracle ne sont donc pas adaptées pour obtenir un verdict partiel.

2.2.4.3 Représentation du résultat de la comparaison de modèles

Une fois le calcul des différences effectué, la question de la représentation du résultat de ce calcul n'est pas un simple point de détail. Il est important de choisir un mode de représentation qui soit adapté pour l'exploitation qui sera faite de ce résultat. Cicchetti et al. [Cicchetti et al., 2007] ont travaillé sur ce sujet et ont énoncé les propriétés attendues pour représenter ces différences, et ont proposé un méta-modèle répondant à leur cahier des charges. Pour eux, le résultat d'une comparaison doit être un modèle et non sous forme textuelle. Ce résultat doit aussi être minimaliste, transformatif, composable. Enfin il doit être indépendant des langages utilisés (et donc des méta-modèles), contrairement à des solutions telles qu'UMLDiff [Xing and Stroulia, 2005] qui est spécifique aux diagrammes UML.

Représentations textuelles Certains outils permettant de calculer des différences, ont fait le choix d'utiliser une représentation textuelle. Ce type de représentation se présente sous la forme d'une séquence d'opérations primitives (ajout, édition, suppression) regroupées dans un fichier texte. Elle possède l'avantage d'être composable, et de pouvoir également faire office de visualisation, ces opérations étant lisibles et compréhensibles par l'utilisateur. Cependant, comme le précisent Cicchetti et al. [Cicchetti et al., 2007] lorsque les fichiers sont optimisés, la compréhension par l'utilisateur est diminuée. Enfin un tel format de fichier se trouve être lié à l'outil duquel il est issu, ce qui risque de limiter l'utilisateur.

Représentations à base de modèles Une représentation à base de modèles est plus en phase avec les principes de l'IDM où tout est modèle. De plus elle offre à l'utilisateur une plus grande flexibilité, lui fournissant de fortes possibilités de réutilisation, par exemple pour réaliser de la détection de conflits ou toute autre analyse ou manipulation. Ce mode de représentation permet également l'intégration dans une chaîne d'outils, à l'image d'EMF Compare [Brun and Pierantonio, 2008] totalement intégré à la plateforme EMF d'Eclipse.

En pratique, les modèles qui sont manipulés dans des cas concrets sont généralement de grande taille, ce qui fait que le calcul de la comparaison peut être long. L'utilisation de comparaisons partielles ou variables peut être une solution pour répondre à ce problème. De plus dans le cas du test d'une transformation de modèles, la transformation n'exploite

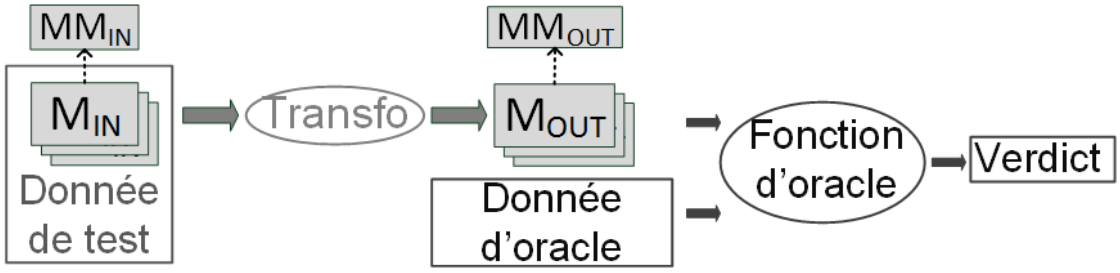


Figure 2.9 – Test d'une transformation de modèles

pas forcément l'ensemble du méta-modèle source. Partant de cette constatation, Sen et al. [Sen et al., 2009] ont mis au point un algorithme pour "élaguer" le méta-modèle afin d'obtenir le méta-modèle effectif de la transformation .

2.3 Test de transformations de modèles

Tester une transformation de modèles c'est lui appliquer le processus présenté Section 2.1.1 (page 15). La Figure 2.9 présente le principe du test d'une transformation de modèles. La donnée de test devient modèle de test ; ce modèle de test est transformé par la transformation sous test en un modèle obtenu. La fonction d'oracle produit un verdict en contrôlant le modèle obtenu à l'aide de la donnée d'oracle.

Le projet DOMINO [3] s'est intéressé à la vérification et la validation dans le domaine de l'IDM [Baudry et al., 2011]. Baudry et al. [Baudry et al., 2010] ont identifié trois défis principaux, repris par Guerra [Guerra, 2012], pour le test de transformations de modèles : (i) la génération de modèles de test, (ii) la définition de critères pour qualifier un ensemble de cas de test, et (iii) la construction de fonctions d'oracle.

Contrairement au *model-based testing* évoqué Section 2.1.2.3 (page 20) nous ne cherchons pas à tester le système représenté par un modèle. Nous nous intéressons à la validation de la transformation de ce modèle en un autre modèle.

2.3.1 Sélection / génération de modèles de test

Le premier défi évoqué par Baudry et al. pour le test de transformations de modèles est la génération de modèles de test.

3. http://www.domino-rntl.org/

Fleurey et al. [Fleurey et al., 2009] proposent de mesurer la couverture du méta-modèle d'entrée. Ils ont adapté les critères définis par Andrews et al. [Andrews et al., 2003] pour la couverture de diagrammes de classe UML :

- `Class Coverage` : chaque méta-classe doit être instanciée au moins une fois.
- `Association End Multiplicities` : pour les extrémités de chaque association, chaque multiplicité représentative doit être couverte.
- `Class Attribute` : pour chaque attribut, chaque valeur représentative intéressante pour le testeur doit être couverte.

Les deux derniers critères de couverture demandent de couvrir des valeurs représentatives. Ces valeurs représentatives sont déterminées à l'aide d'une analyse partitionnelle. Le domaine de valeur d'un attribut ou d'une multiplicité est partionné en classes d'équivalences au sein desquelles le comportement de la transformation est supposé identique. Une valeur est choisie par classe d'équivalence.

Fleurey et al. combinent ensuite ces valeurs représentatives pour former des fragments de modèles. Chacun de ces fragments de modèles doit être inclus dans au moins un modèle de test. Si tous les fragments ne sont pas couverts ils ajoutent les modèles de test nécessaires. Sen et al. [Sen et al., 2012] ont développé un outil reposant sur ces critères pour générer automatiquement un ensemble de modèles de test.

Bauer et al. [Bauer and Küster, 2011] testent des chaînes de transformations de modèles. Ils combinent la couverture du code et la spécification des transformations. Gonzales et al. [González and Cabot, 2012] proposent une approche de génération de modèles de test en boîte blanche, dédiée aux transformations ATL. McQuillan et al. [McQuillan and Power, 2009] s'intéressent eux aussi aux techniques de générations de modèles de test en boîte blanche pour les transformations ATL.

2.3.2 Oracle du test de transformations de modèles

La construction de fonctions d'oracles est un deuxième défi identifié par Baudry et al. pour le test de transformations de modèles. Pour relever ce défi, Mottu et al. [Mottu et al., 2008] proposent plusieurs fonctions d'oracles destinées au test de transformations de modèles. Ces fonctions d'oracles s'articulent autour de deux axes principaux : l'utilisation de modèles attendus et celle de contrats ou d'assertions. Selim et al. [Selim et al., 2012] identifient eux aussi ces deux axes. Le testeur a donc à sa disposition un ensemble de fonctions d'oracles, mais nous pouvons nous demander si cet ensemble de fonctions d'oracles permet de répondre à tous ses besoins (Question **A**, page 2).

2.3.2.1 Oracles utilisant des modèles attendus

Plusieurs des fonctions d'oracles proposées par Mottu et al. sont inspirées de la plus simple présentée Section 2.1.2 (page 16), la comparaison du résultat obtenu avec le résultat attendu. Kolovos et al. [Kolovos et al., 2006] reconnaissent également que la comparaison de modèles est une composante majeure du test de transformations de modèles. Dans ce cas, le modèle obtenu après exécution de la transformation sous test est comparé à un modèle correspondant au résultat attendu pour la transformation correcte du modèle de test. Le test passe si les deux modèles sont identiques.

La principale difficulté d'une telle fonction d'oracle est l'obtention du modèle attendu. Un modèle attendu est associé à chaque modèle de test. Le testeur crée manuellement ce modèle attendu ou il l'obtient.

Une solution proposée pour éviter que le testeur écrive tous les modèles attendus est d'utiliser une *transformation de référence*. Cette transformation de référence est une autre mise en œuvre de la transformation sous test que le testeur sait être valide. Pour chaque cas de test, le modèle de sortie produit par la transformation de référence est utilisé comme modèle attendu et est comparé au modèle produit par la transformation sous test. Cette solution évite au testeur d'avoir à créer manuellement les modèles attendus, mais son utilisation est limitée. Un cas de figure où cette fonction d'oracle peut être utilisée est le cas où la transformation sous test est une version optimisée d'une transformation déjà mise en œuvre. Cependant ce cas de figure reste peu répandu. La majorité des transformations développées ne l'ont pas été auparavant ou sont des mises à jour de transformations existantes. Elles ne peuvent donc pas être utilisées comme mise en œuvre de référence pour l'oracle. De plus, même si une telle mise en œuvre validée est disponible, elle n'est pas toujours utilisable pour l'oracle.

Une autre solution est envisagée pour ne pas avoir à créer les modèles attendus manuellement. Cette solution repose sur la notion de transformation bidirectionnelle [Stevens, 2008, Czarnecki et al., 2009]. L'idée est que dans certains cas, pour une transformation T qui transforme un modèle d'entrée M en un modèle de sortie N, il existe une *transformation inverse* T' qui prend en entrée le modèle N et produit en sortie le modèle M. Ainsi, Mottu et al. proposent de transformer le modèle obtenu après exécution de la transformation sous test, à l'aide de la transformation inverse. Le modèle de sortie produit par la transformation inverse est comparé au modèle de test. Encore une fois, le test passe si les modèles sont identiques. Malheureusement l'utilisabilité de cette fonction d'oracle est également limitée. D'une part, d'après Stevens [Stevens, 2008], seules les transformations bijectives peuvent être bidirectionnelles, il est nécessaire que les modèles d'entrée et de sortie soient équivalents, qu'ils contiennent les mêmes informations. S'il y a une perte d'information lors du passage du modèle M au modèle N alors il n'est pas possible d'obtenir le modèle M à partir du modèle N. D'autre part, il ne suffit pas que la transformation sous test soit bijective pour pouvoir utiliser cette fonction d'oracle. Il faut

également que le testeur ait à sa disposition une mise en œuvre valide de la transformation inverse ; ce qui limite encore les cas d'utilisation possibles.

De même Lin et al. [Lin et al., 2005] proposent un environnement dédié au test de transformations de modèles, utilisant la comparaison de modèles comme fonction d'oracle. D'un point de vue technique, ils utilisent le langage ECL (Epsilon Comparison Language) pour effectuer la comparaison. L'inconvénient d'utiliser un langage spécifique comme ECL pour la comparaison oblige le testeur à apprendre et maîtriser ce langage qu'il ne connaît pas forcément.

2.3.2.2 Oracles utilisant des contrats

D'autres fonctions d'oracle partent du principe qu'un modèle de sortie correct pour la transformation sous test doit respecter certaines contraintes.

L'utilisation de contrats pour contrôler le résultat de l'exécution d'un programme a été proposée par Meyer [Meyer, 1992]. Nous l'avons vu, un contrat exprime des contraintes sur le résultat produit par le système sous test ou sur les relations entre entrée et sortie (Section 2.1.2 page 16). Beugnard et al. [Beugnard et al., 1999] ont établi une classification des différents types de contrats pour des composants. Chaque type de contrat correspond à un niveau de vérification.

Cariou et al. [Cariou et al., 2009], proposent d'utiliser des contrats génériques écrits en OCL pour valider des transformations de modèles. Avec ces contrats ils vérifient trois types de contraintes :

- des contraintes sur le modèle source : un ensemble de pré-conditions qu'un modèle d'entrée correct, d'après la spécification de la transformation sous test, doit respecter.
- des contraintes sur le modèle cible : un ensemble de post-conditions générales qu'un modèle obtenu correct doit respecter indépendamment du modèle d'entrée. Par exemple, pour la transformation de mise à plat d'une machine à état, le modèle de sortie ne doit pas contenir d'états composites.
- des contraintes sur l'évolution des éléments du modèle d'entrée : ces contraintes sont également un ensemble de post-conditions qu'un modèle obtenu correct doit respecter. À la différence des précédentes contraintes sur le modèle cible, ici les contrats décrivent des relations existantes entre le modèle d'entrée et le modèle de sortie. Si elles sont respectées, alors le modèle obtenu est le résultat d'une transformation correcte du modèle de test. Par exemple, après mise à plat d'une machine à état, les états simples présents dans le modèle de test doivent également être présents dans le modèle de sortie.

Nativement, OCL ne permet pas d'écrire des contraintes portant sur plusieurs modèles. C'est pourquoi Cariou et al. proposent de concaténer le modèle de test avec le modèle obtenu.

Si Cariou et al. concatènent leurs modèles pour une raison technique, d'autres approches proposent de modéliser la transformation sous test pour la valider. Braga et al. [Braga et al., 2011, de O. Braga et al., 2012], introduisent la notion de méta-modèle de transformation. La transformation sous test est représentée sous la forme d'un méta-modèle, et une instance de ce méta-modèle est une exécution de la transformation sous test. Pour valider la transformation, ils écrivent des contrats pour vérifier des propriétés au niveau du méta-modèle. Ces contrats sont des invariants qui doivent être valides au niveau modèle.

Esther Guerra [Guerra, 2012] a développé *TransML*, une famille de langages de modélisation dédiée aux transformations de modèles. Ils génèrent à la fois les données de test et les oracles en s'appuyant sur une spécification de la transformation sous test en *TransML*. Cette spécification peut être considérée comme un contrat.

L'utilisation de contrats génériques comme fonction d'oracle possède un avantage par rapport à la comparaison de modèles. Dans le second cas le testeur doit fournir un modèle attendu spécifique à chaque cas de test, alors que les contrats génériques sont spécifiques à la transformation sous test. Ils sont définis une seule fois pour tous les cas de test. L'inconvénient de l'utilisation de contrats comme fonction d'oracle est leur complexité.

Pour Mottu et al. [Mottu et al., 2008], les contrats écrits par le testeur pour contrôler entièrement un modèle de sortie sont aussi complexes que la transformation sous test en elle-même, le risque d'écrire des contrats erronés est aussi élevé que pour la transformation sous test.

Gogolla et al. [Gogolla and Vallecillo, 2011] proposent d'utiliser des *tracts*. Les *tracts* sont des contrats partiels (sur le modèle d'entrée, de sortie, ou les relations entre les deux). Ces contrats partiels, sont plus simples que des contrats génériques, le risque d'erreur est donc moins important.

Une autre fonction d'oracle proche des contrats utilise des assertions. Ces assertions, généralement écrites en OCL contrôlent des propriétés individuelles du modèle obtenu. À la différence d'un contrat générique, une assertion est spécifique à un modèle obtenu et L'assertion utilisée peut aussi prendre la forme d'un *pattern*. La fonction d'oracle vérifie que le modèle obtenu contient le *pattern*. Le *pattern* permet d'exprimer des propriétés attendues dans le modèle de sortie. Ces *patterns* peuvent être exprimés à l'aide de *model snippets* [Ramos et al., 2007], des morceaux de modèle. Tiso et al. [Tiso et al., 2012] s'intéressent aux transformations modèle vers texte. Ils utilisent des assertions pour contrôler des propriétés de la sortie produite. Selon eux, l'écriture des assertions devrait se faire en boîte blanche, ils doivent tenir compte de choix faits par le développeur de la transformation.

Des assertions sont également plus simples que des contrats génériques mais elles doivent être définies pour chaque cas de test. De la même manière qu'un modèle n'est pas uniquement un ensemble d'éléments (sa structure est importante), nous souhaitons valider le modèle obtenu dans sa globalité et pas uniquement un ensemble de propriétés.

2.3.2.3 Test de transformations de modèles sans oracle

La définition d'un oracle est l'un des principaux défis du test de transformations de modèles. C'est un processus réalisé essentiellement manuellement par le testeur. A l'heure actuelle il n'existe aucun moyen d'automatiser la création d'un oracle. C'est pourquoi des travaux ont été entrepris pour tenter de tester une transformation de modèles sans avoir besoin d'un oracle.

Kessentini et al. [Kessentini et al., 2011] proposent une approche dans ce sens. Ils utilisent comme référence un ensemble d'exemples de modèles d'entrée et les modèles de sortie leur correspondant ainsi que les liens de traçabilité entre ces modèles. Ils expriment les modèles et les liens de traçabilité à l'aide de prédicats. Ils utilisent ensuite un algorithme génétique pour générer à partir de leurs exemples des traces de transformations erronées ou détecteurs. Enfin après exécution de la transformation sous test sur les modèles de test, ils comparent les traces d'exécution à leur détecteurs. Les traces obtenues pour les transformations sous test qui sont proches d'un de leurs détecteurs indiquent un risque d'erreur dans la transformation sous test. Ces risques d'erreurs sont classés selon leur proximité avec les détecteurs ; le testeur peut ensuite analyser ces résultats et vérifier s'il s'agit de réelles fautes dans la transformation. Le principal problème de cette approche vient de la base d'exemples. Il n'est pas évident que le testeur puisse avoir à sa disposition des exemples de modèles d'entrée et de sortie suffisamment proches pour être utiles. S'il n'a pas ces modèles à sa disposition, alors il lui faudra les définir par lui-même, ce qui reviendrait à définir des oracles pour un ensemble de données de test.

2.3.2.4 Fonctions d'oracle existantes et verdict partiel

Le rôle de l'oracle d'un cas de test est de contrôler le modèle obtenu et de produire un verdict. Si l'oracle ne contrôle qu'une partie du modèle obtenu ou ne prend en compte qu'une partie de la spécification de la transformation sous test, alors il produit un verdict partiel. Dans cette sous-section nous discutons de l'utilisation des fonctions d'oracles existantes pour obtenir un verdict partiel. Nous nous plaçons dans la situation où le testeur ne s'intéresse qu'à une partie du modèle obtenu. Nous appelons *partie contrôlée* la partie du modèle obtenu à laquelle le testeur s'intéresse. Le reste du modèle est la *partie non contrôlée*.

Nous avons présenté ci-dessus les différentes fonctions d'oracle existantes pour le test de transformations de modèles. Nous avons distingué deux catégories de fonctions d'oracle :

- celles qui utilisent une comparaison de modèle ;
- celles qui vérifient des propriétés du modèle obtenu.

Les fonctions d'oracle de la première catégorie utilisent une comparaison de modèles. La plus simple des fonctions d'oracle de cette catégorie compare le modèle obtenu à un modèle attendu créé par le testeur. Pour obtenir un verdict partiel le testeur peut vouloir

comparer le modèle obtenu à un modèle attendu partiel. Ce modèle attendu partiel serait un modèle qui contient uniquement la valeur attendue pour la partie contrôlée. Dans ce cas, la comparaison de modèle n'est pas une fonction d'oracle adaptée. Comparer le modèle obtenu à un modèle attendu partiel créerait un faux positif. Même si la partie contrôlée du modèle obtenu est identique au modèle attendu partiel, des différences seront observées pour la partie non contrôlée. Il n'est pas actuellement possible de ne comparer qu'en partie deux modèles distincts.

L'utilisation d'une mise en œuvre de référence de la transformation sous test repose aussi sur une comparaison de modèles. Cette mise en œuvre de référence fournit au testeur un modèle attendu entier ; avec ce modèle attendu, le testeur peut entièrement contrôler le modèle obtenu. Dans ce cas, le testeur n'a pas besoin d'un verdict partiel. Par contre, si la mise en œuvre de référence ne correspond pas à la même spécification, mais en est proche (par exemple s'il s'agit d'une version précédente de la transformation sous test) alors un verdict partiel serait intéressant. Le testeur peut vouloir utiliser la version précédente comme mise en œuvre de référence pour un test de non régression. De nouveau, nous souhaitons obtenir un verdict partiel à partir d'une comparaison du modèle obtenu avec un modèle attendu partiel.

L'utilisation d'une transformation inverse est la dernière fonction d'oracle qui repose sur une comparaison de modèles. Là encore si le testeur possède une telle transformation, alors un verdict partiel ne lui est d'aucune utilité, il est capable de valider entièrement le modèle obtenu.

Les oracles de la seconde catégorie contrôlent des propriétés du modèle obtenu. Les contrats génériques, proposés par Cariou et al. [Cariou et al., 2009] pour le test de transformations de modèles appartiennent à cette catégorie. Ces contrats ne sont pas spécifiques à un cas de test, ils sont définis au niveau de la transformation sous test. Les propriétés contrôlées dans le modèle de sortie sont exprimées par rapport au modèle d'entrée. Nous l'avons vu dans la Section 2.3.2.2, écrire des contrats est au moins aussi complexe qu'écrire la transformation. Donc il y a autant de chances d'avoir des contrats erronés que de trouver des fautes dans la transformation sous test.

Ces contrats génériques non plus, ne sont pas adaptés à notre besoin. Ils sont seulement utilisables pour contrôler une partie de la spécification, par exemple l'absence d'états composites. Vallecillo et al. [Vallecillo et al., 2012] en viennent à la même conclusion et proposent les *Tracts*, des contrats partiels dédiés à cette utilisation. Des contrats partiels sont intéressants parce qu'ils sont simples et donc peu sujets à erreur. La partie contrôlée du modèle de sortie n'est pas forcément petite, elle peut contenir la majorité des éléments du modèle. Ainsi, pour notre besoin, ces contrats partiels ne sont pas adaptés. Par contre ils pourraient être utilisés en complément d'autres fonctions d'oracles, pour compléter un verdict partiel.

D'autres fonctions d'oracle qui appartiennent à la seconde catégorie utilisent des *assertions ou patterns* (i.e., des contraintes OCL sur le modèle obtenu d'un cas de test). Ces

fonctions permettent de contrôler des propriétés individuelles sur les modèles. Un modèle n'est pas qu'un ensemble de propriétés individuelles, l'organisation de ces propriétés, leur structure est importante également. Notre but n'est pas de contrôler un sous-ensemble des propriétés du modèle obtenu, nous souhaitons contrôler entièrement une partie de ce modèle. L'utilisation d'assertions ou de *patterns* n'est donc pas non plus adaptée à nos besoins.

Les techniques de test sans oracles ne sont pas non plus une solution pour obtenir un verdict partiel à propos d'une partie du modèle obtenu. Celle de Kessentini et al. [Kessentini et al., 2011] utilise une base d'exemples composée de modèles d'entrée associés au modèle de sortie qui leur correspond. S'il n'y a pas de base d'exemples adéquate le testeur va devoir la créer, ce qui revient à créer des cas de test. De plus si la transformation a des sorties polymorphes, alors les exemples existants ne reflètent pas forcément le choix qui a été fait par le développeur. Il y a alors un risque de faux positifs.

2.3.3 Qualité des tests

L'évaluation de la qualité de cas de test de transformations de modèles est le dernier défi évoqué par Baudry et al. [Baudry et al., 2010]. Nous nous intéressons, dans un premier temps, aux approches pour évaluer la qualité des modèles de test. Dans un second temps, nous considérons l'évaluation de la qualité des oracles pour le test de transformations de modèles.

2.3.3.1 Qualification des modèles de test

Dans le contexte du test de transformations de modèles, des techniques de mesure de couverture et l'analyse de mutation sont utilisées pour qualifier des données de test.

Il existe deux approches différentes pour la mesure de couverture pour des transformations de modèles [Bauer and Küster, 2011] : D'un côté, le testeur peut mesurer la couverture du code de la transformation ; d'un autre côté, il est également possible d'évaluer le taux de couverture de la spécification de la transformation sous test par les cas de test.

McQuillan et al. [McQuillan and Power, 2009] proposent des critères de couverture du code pour une transformation écrite en ATL. Un de ces critères est la couverture des règles de la transformation.

Les modèles manipulés par une transformation sont conformes à des méta-modèles. Ces méta-modèles font partie de la spécification d'une transformation. Évaluer la couverture de la spécification par les cas de test, peut se faire en évaluant la couverture des méta-modèles.

Les travaux traitant de la couverture d'un méta-modèle par un modèle concernent la génération ou la sélection de modèles de test pour le test d'une transforma-

tion [Wu et al., 2012]. Dans ces travaux, le but est d'évaluer la couverture du modèle d'entrée par les modèles de test. Andrews et al. [Andrews et al., 2003] ont défini des critères de couverture pour des diagrammes de classes UML. Constatant qu'un méta-modèle est proche d'un diagramme de classes, Fleurey et al. [Fleurey et al., 2009] ont adapté ces critères pour la couverture de méta-modèles.

Mottu et al. [Mottu et al., 2006a] proposent d'utiliser l'analyse de mutation pour qualifier un ensemble de modèles de test. Le testeur crée des mutants en introduisant des fautes dans la transformation sous test. Une seule faute est ajoutée à la transformation sous test pour créer chaque mutant. Des opérateurs de mutation décrivent les fautes à ajouter ; Mottu et al. ont identifié les fautes susceptibles d'être présentes dans la mise en œuvre d'une transformation de modèles et ont défini des opérateurs de mutation spécifiques. Ces opérateurs sont indépendants du langage de transformation utilisé.

Le processus de l'analyse de mutation pour qualifier un ensemble de modèles de test est décrit dans la Figure 2.10, il est le suivant :

1. le testeur crée des modèles de test ;
2. le testeur transforme les modèles de test à l'aide de la transformation sous test ;
3. le testeur applique les opérateurs de mutation pour créer les mutants ;
4. le testeur transforme les modèles de test à l'aide des mutants ;
5. pour chaque mutant le testeur compare les modèles obtenus avec ceux produits par la transformation sous test ;
6. si au moins un modèle est différent, le mutant est tué, sinon il est vivant ;
7. parmi les mutants vivants, le testeur élimine ceux qui sont équivalents à la transformation sous test ;
8. le testeur calcule le score de mutation : le rapport du nombre de mutants tués sur le nombre total de mutants non équivalant ;
9. si le score de mutation est satisfaisant, alors l'ensemble de modèles de test est lui aussi satisfaisant ;
10. sinon, le testeur ajout de nouveaux modèles de test et recommence jusqu'à obtenir un score de mutation satisfaisant.

Aranega et al. [Aranega et al., 2011a] ont ajouté la traçabilité à l'analyse de mutation pour aider le testeur à améliorer les données d'entrée.

Khan et al. [Khan and Hassine, 2013] utilisent eux aussi l'analyse de mutation pour qualifier des modèles de test pour une transformation ATL. Ils définissent 10 opérateurs de mutation dédiés au langage ATL, certains sont des adaptations des opérateurs génériques définis par Mottu et al. [Mottu et al., 2006a] d'autres sont spécifiques aux mécanismes d'ATL. A l'aide de ces opérateurs ils ont créés des mutants pour une transformation et ont évalué leurs cas de test.

Si l'analyse de mutation donne des résultats satisfaisants pour qualifier des données d'entrée, elle a plusieurs inconvénients. Le premier inconvénient de l'analyse de mutation

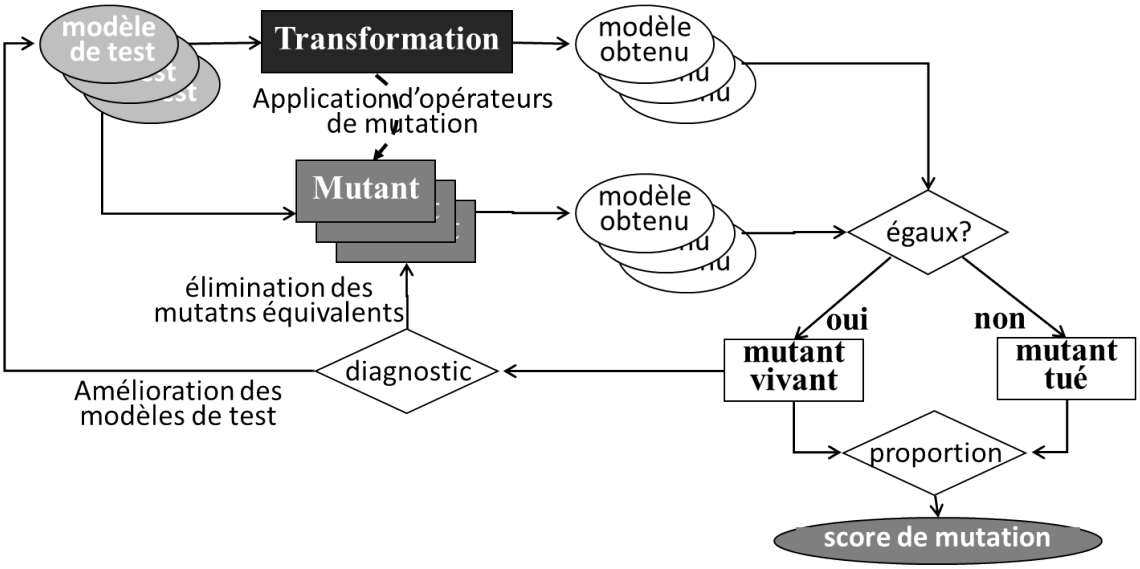


Figure 2.10 – Utilisation de l'analyse de mutation pour qualifier des modèles de test

est qu'elle est dépendante du langage de transformation utilisé. Le testeur crée des mutants de la transformation sous test en appliquant des opérateurs de mutation. Un opérateur de mutation décrit une faute susceptible d'être présente dans la transformation sous test. Mottu et al. [Mottu et al., 2006a] ont proposé des opérateurs de mutation spécifiques aux transformations de modèles. Ces opérateurs de mutation sont définis indépendamment d'un langage de mutation. Cependant, l'application d'un opérateur à une transformation varie en fonction du langage de transformation utilisé.

Le deuxième inconvénient de l'analyse de mutation est qu'une partie des tâches est effectuée manuellement par le testeur. L'application des opérateurs de mutation est dépendante du langage utilisé. Pour automatiser cette tâche il faudrait prendre en compte les spécificités de chaque langage. Cette tâche est donc effectuée par le testeur.

Le troisième inconvénient de l'analyse de mutation est une conséquence des deux précédents ; il s'agit de son coût. Suivant la complexité de la transformation sous test, le testeur peut créer un grand nombre de mutants. La création manuelle de tous ces mutants par le testeur représente tout d'abord un effort et un temps non négligeable. Ensuite, chaque mutant est exécuté sur chaque modèle de test, et chaque modèle obtenu est contrôlé par un oracle.

2.3.3.2 Qualification d'oracles

En complément de son utilisation pour qualifier des modèles de test, Mottu et al. [Mottu et al., 2006b] ont également étudié l'application de l'analyse de mutation pour

la qualification de l'oracle du test de transformations de modèles. Ils appliquent l'approche proposée par Jezequel et al. [Jézéquel et al., 2001].

Cette application suit les étapes suivantes (illustrée dans la Figure 2.11) :

1. ils utilisent l'analyse de mutation pour obtenir un ensemble de modèles de test satisfaisant (voir Section 2.3.3.1, page 39) ;
2. ils conservent uniquement les mutants tués par les modèles de test, et leurs modèles de sortie obtenus ;
3. ils créent les oracles qui correspondent à ces modèles de test ;
4. ils contrôlent les modèles produits par la transformation sous test ;
5. si tous les tests ne passent pas, alors ils corrigent leurs oracles jusqu'à ce qu'il n'y ait plus aucun échec ;
6. si tous les tests passent, ils contrôlent les modèles obtenus de chaque mutant avec leurs oracles ;
7. un mutant est tué si une erreur est détectée dans un de ses modèles attendus ;
8. ils calculent le nouveau score de mutation (nombre de mutants tués / nombre total de mutants) ;
9. si le score de mutation n'est pas satisfaisant ils améliorent leurs oracles, et recommencent le processus ;
10. si le score de mutation est satisfaisant, alors les oracles sont satisfaisants ;

Nous avons déjà évoqué les défauts de l'analyse de mutation (dépendante du langage de transformation, donc mise en application manuelle et coûteuse, mais son utilisation pour l'évaluation de la qualité des oracles en apporte un supplémentaire. Cet inconvénient concerne l'amélioration des oracles qualifiés. Si des mutants sont encore en vie, alors les oracles doivent être améliorés. Le testeur sait dans quelle instruction de la transformation une faute a été introduite. Pour améliorer les oracles, il devra associer cette instruction à un élément du modèle obtenu ou une partie de la spécification de la transformation sous test. Pour améliorer les oracles il faudra contrôler cet élément ou cette partie de la spécification. Malheureusement ce lien est difficile à établir [Aranega et al., 2011b].

Esther Guerra [Guerra et al., 2013] utilise également des contrats pour l'oracle du test de transformations de modèles. Plutôt que d'utiliser OCL, elle propose TransML, un langage dédié ayant une sémantique formelle. Du fait de cette sémantique formelle, elle évoque la possibilité de raisonner sur la qualité des contrats selon plusieurs critères :

- La couverture des méta-modèles d'entrée et de sortie, pour éviter la sous-spécification.
- Les redondances dans les contrats. Si la pré ou post-condition d'un contrat est inclue dans un autre contrat plus important. Le plus petit contrat est inutile et peut être supprimé.
- Les contradictions entre deux contrats.

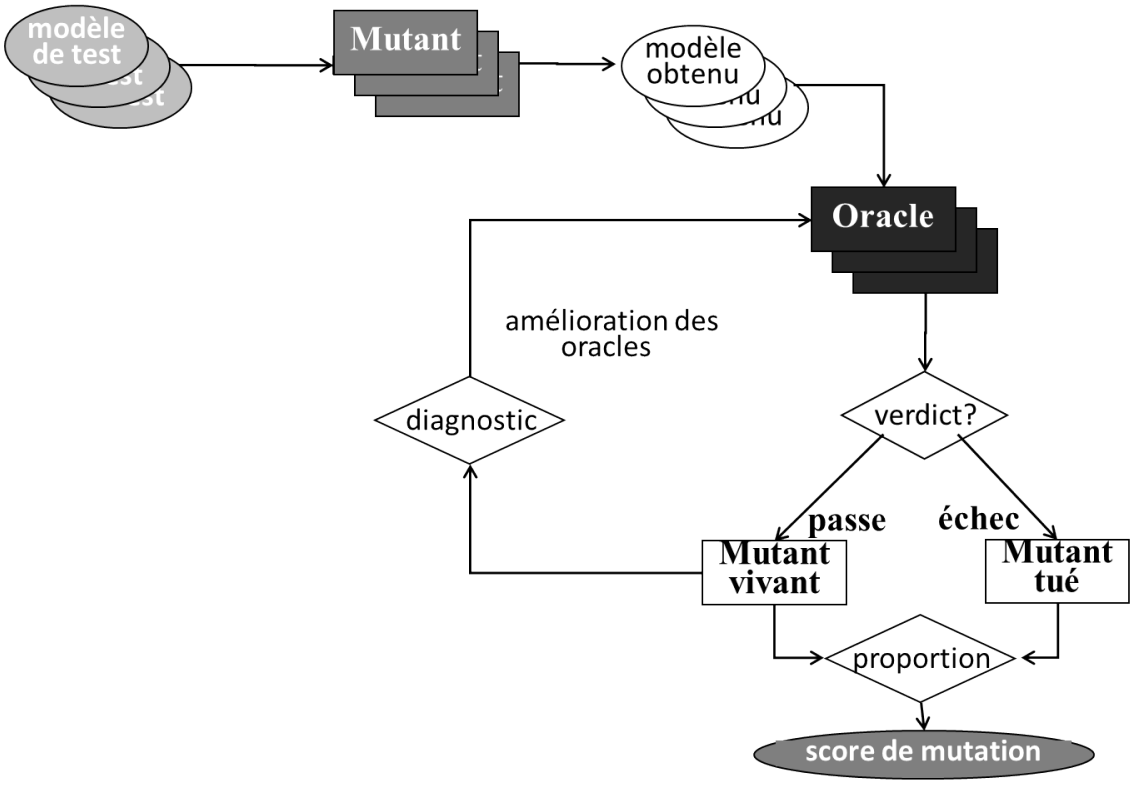


Figure 2.11 – Utilisation de l'analyse de mutation pour qualifier un ensemble d'oracles

– La satisfaction de patterns ; par exemple une pré-condition qui rendrait inutile un invariant.

Cette approche parait intéressante pour qualifier des oracles, mais elle est spécifique à leur langage que le testeur devra apprendre. De plus, elle ne précise pas comment elle compte mesurer la couverture des méta-modèles et sur quels critères elle se baserait pour contrôler ces propriétés.

2.3.4 Utilisation de la traçabilté pour le test de transformations de modèles

La notion de traçabilité a été introduite dans le domaine par Winkler et al. [Winkler and Pilgrim, 2010]. Ils la décrivent comme la possibilité de suivre la vie d'artefacts logiciels. Dans le contexte de l'IDM, ils citent la définition donnée par l'OMG : "Une trace enregistre un lien entre un groupe d'objets dans les modèles d'entrée et un groupe d'objets dans les modèles de sortie. Ce lien est associé à un élément de la spécification de la transformation qui relie les groupes concernés."

Amar et al. [Amar et al., 2008] proposent un moteur de traçabilité pour des transformations de modèles, qui aboutit à la création d'une trace représentée à l'aide d'un graphe. Durant une exécution de la transformation tous les évènements affectant le modèle donnent lieu à la création d'un lien de traçabilité. Ainsi, en fin de processus, le graphe obtenu permet de visualiser la transformation, chaque élément du modèle de sortie est lié à un élément du modèle d'entrée ainsi qu'à la règle qui a amené à sa production. Le moteur est disponible sous la forme d'un plugin Eclipse. Il est utilisable avec une transformation écrite en Java/EMF.

Jouault et al. [Jouault, 2005] proposent d'ajouter des informations de traçabilité pour des transformations mises en œuvre en ATL. Ils considèrent que les informations de traçabilité sont un modèle, plus précisément les traces observées lors de l'exécution sont fournies dans un modèle de sortie additionnel pour la transformation. ATL supporte nativement la traçabilité, mais pour l'intégrer à une transformation il est nécessaire de décorer son code ATL pour y ajouter des instructions spécifiques.

Aranega et al. [Aranega et al., 2008] présentent quant à eux une utilisation de la traçabilité destinée à la localisation d'erreurs dans une transformation. Leur approche consiste aussi à construire un modèle de trace pour une transformation de modèles. Cette trace relie chacun des éléments du modèle obtenu à la séquence de règles dont il est issu. Une fois la transformation effectuée, l'exécution de tests sur le modèle de sortie permet de détecter d'éventuelles erreurs. Le but du travail présenté est, en utilisant la trace obtenue, de se rapprocher des règles incorrectes qui ont produit les éléments à l'origine des éventuelles erreurs détectées.

2.3.5 Vérification de transformations de modèles

En plus du test, d'autres techniques de génie logiciel peuvent être utilisées pour valider une transformation de modèles. Certains auteurs ont notamment étudié la possibilité de vérifier formellement une transformation de modèles. Un modèle peut être vu comme un graphe, certaines des approches présentées ici s'appuient sur ce point de vue et utilisent des résultats existants pour l'analyse de transformations de graphes.

2.3.5.1 Propriétés d'une transformation de modèles

La vérification d'une transformation de modèles repose sur le fait de contrôler la satisfaction de propriétés par la transformation. D'après Hermann et al. [Hermann et al., 2010], ces propriétés sont principalement la terminaison, la correction, et la complétude.

Une autre propriété que doit satisfaire une transformation est la confluence ; Ehrig et al. [Ehrig et al., 2010a] décrivent la confluence comme une propriété assurant le comportement fonctionnel de la transformation. Selon eux, un système de transformations de

graphes est confluant si lorsque qu'un graphe K peut être transformé en deux graphes distincts P_1 et P_2, alors chacun de ces graphes peut être transformé en un même graphe K'. Cela signifie que pour chaque graphe dans le langage source (ou chaque modèle d'entrée), la transformation produira un unique graphe dans le langage cible (ou un unique modèle de sortie). Nous avons évoqué le cas des transformations de modèles aux sorties polymorphes, où plusieurs variantes existent pour un modèle de sortie donné. Ce cas particulier ne viole pas cette propriété de confluence ; il ne s'agit pas de plusieurs modèles distincts mais bien de plusieurs variantes d'un même modèle de sortie exprimant une même sémantique.

Küster et al. [Küster and Abd-El-Razik, 2006] dressent une liste des erreurs qui peuvent apparaître lors de l'écriture d'une règle de transformation. Outre les classiques erreurs de codage, une règle peut aussi ne prendre en compte qu'une partie des éléments du modèle d'entrée qu'elle est sensée transformer. Elle peut également produire des modèles de sortie incorrects d'un point de vue syntaxique (non conformes au méta-modèle de sortie), ou sémantique (conforme au méta-modèle de sortie mais n'étant pas une transformation correcte du modèle d'entrée). Enfin le problème de la confluence est évoqué, la transformation peut produire plusieurs modèles de sortie différents pour un même modèle d'entrée. Küster [Küster, 2006] a également proposé une approche pour définir des transformations de modèles, de manière à ce qu'elles terminent, soient syntaxiquement correctes et aussi confluentes.

Amrani et al. [Amrani et al., 2012] ont réalisé une étude des différentes approches existantes pour la vérification des transformations de modèles. Les approches qu'ils présentent vérifient des propriétés aussi bien liées au langage utilisé pour mettre en œuvre la transformation que sur les modèles qu'elle manipule. En ce qui concerne les propriétés liées au langage de transformation utilisé, ils citent : la terminaison de la mise en œuvre, son déterminisme, et son typage. Ils rapprochent le déterminisme de la mise en œuvre d'une transformation de la notion de confluence mentionnée ci-dessus. Quant au typage, il s'agit de s'assurer qu'il n'y a pas d'erreurs de syntaxe dans la mise en œuvre. Pour les auteurs, cette propriété vise plus particulièrement les langages de modélisation visuelle, tels que celui proposé par Guerra et al. [Guerra et al., 2013]. En effet les langages textuels comme Kermeta ou ATL bénéficient de l'expérience accumulée pour créer des Environnements de Développement Intégrés pour les langages de programmation généraux.

Pour ce qui est des vérifications liées aux modèles, les auteurs précisent qu'il est possible de vérifier des propriétés des modèles d'entrée et/ou de sortie (telles que la conformance à un méta-modèle), d'autres propriétés liées à la syntaxe ou encore la sémantique des modèles. La vérification de propriétés liées à la syntaxe des modèles s'intéresse aux relations structurelles entre modèles d'entrée et de sortie. Les contrats présentés dans la Section 2.3.2.2 se rapprochent de ce type de vérification. Les dernières propriétés vérifiées concernent la sémantique des modèles. Le but ici est de vérifier que les modèles d'entrée et de sortie expriment, au moins en partie une même sémantique. Par exemple, il peut s'agir de vérifier que le modèle de sortie de la Figure 2.7 (page 27) produit par la

transformation de mise à plat représente le même comportement que le modèle d'entrée de la Figure 2.3 (page 25).

2.3.5.2 Approches existantes

Dans leur étude, Amrani et al. [Amrani et al., 2012] ont classé les techniques de vérification selon deux critères : leur indépendance vis-à-vis de la transformation et vis-à-vis du modèle d'entrée.

Les techniques de la première catégorie sont indépendantes de la transformation et du modèle d'entrée ; elles permettent de vérifier certaines propriétés pour toutes les transformations écrites dans un langage donné. Ces techniques permettent essentiellement de prouver la terminaison ou la confluence d'une transformation, comme l'approche proposée par Küster [Küster, 2006].

Les techniques de la deuxième catégorie sont les plus nombreuses. Elles sont dépendantes de la transformation, mais indépendantes du modèle d'entrée. Elles visent à vérifier des propriétés de la transformation pour n'importe quel modèle d'entrée. Braga et al. [de O. Braga et al., 2012] modélisent une transformation à l'aide d'un méta-modèle. Toute exécution de la transformation est une instance de ce méta-modèle. Pour vérifier cette transformation, ils utilisent des contrats exprimés à l'aide d'un méta-modèle de transformation accompagné d'un ensemble de contraintes sur ce méta-modèle. D'une manière similaire, Büttner et al. [Büttner et al., 2011] modélisent une transformation de modèles pour la vérifier. Ils regroupent au sein d'un même modèle tous les éléments des méta-modèles d'entrée et de sortie ainsi que de la mise en œuvre ATL de la transformation. Ils valident ensuite ce modèle. Cabot et al. [Cabot et al., 2010] modélisent également les transformations pour les valider et ajoutent un invariant à ce modèle de la transformation. Le modèle de transformation est automatiquement généré à partir d'une mise en œuvre déclarative de la transformation.

Giese et al. [Giese et al., 2006] utilisent du *model checking* pour vérifier une transformation spécifiée dans une grammaire de graphes. Lucio et al. [Lúcio et al., 2010] eux aussi valident une transformation de modèles à l'aide de techniques de model checking pour vérifier une transformation spécifiée avec l'outil DSLTrans qu'ils ont développé.

Un autre moyen pour vérifier une transformation de modèles, est de la prouver ; Schätz [Schätz, 2010] utilisent l'assistant de preuve Isabelle. Dans leur approche, les modèles sont formalisés et la transformation est mise en œuvre en règles Prolog.

Enfin les approches restantes sont-elles spécifiques à la transformation d'un modèle d'entrée donné, elles utilisent généralement la traçabilité. Pour Narayanan et al. [Narayanan and Karsai, 2008b], si le modèle de sortie produit après exécution de la transformation est correct, alors il existe un lien vérifiable entre ce modèle obtenu et le modèle d'entrée. Ils proposent ainsi les spécifications d'un langage dédié à cette tâche [Narayanan and Karsai, 2008a].

Comme nous l'avons vu, il existe plusieurs approches pour la vérification formelle d'une transformation de modèles. Cependant, pour Gogolla et al. [Gogolla and Vallecillo, 2011] ces méthodes formelles ne sont pas appropriées pour valider entièrement des transformations de modèles de grande taille du fait de leur complexité.

2.4 Synthèse

Si la génération et la sélection de modèles de test est une tâche automatisable et automatisée, ce n'est pas toujours le cas pour l'oracle. La création d'un oracle est réalisée par le testeur d'après la spécification de la transformation sous test. Si la spécification est formelle et que l'on peut la traiter automatiquement, alors dans ce cas, à l'image du model-based testing il est possible de générer l'oracle à partir de cette spécification. Cependant, la spécification d'une transformation est plus souvent une documentation textuelle que quelque chose que l'on puisse exploiter automatiquement. Un oracle est donc le plus souvent créé manuellement par le testeur. Notre but est ainsi d'assister le testeur dans cette tâche manuelle.

D'une part, le testeur ne souhaite parfois contrôler qu'une partie du modèle obtenu en utilisant un modèle attendu partiel. Cependant aucune des fonctions d'oracle existantes n'est adaptée à ce besoin. L'une de ces fonctions d'oracles est la comparaison du modèle obtenu avec un modèle attendu. Avec cette fonction d'oracle le modèle attendu doit être entier sinon la comparaison produirait un faux positif. Il n'est pas non plus possible d'utiliser des contrats génériques qui sont trop complexes et donc trop susceptibles d'être erronés.

D'autre part, la qualité des oracles des cas de tests est une mesure utile pour le testeur. L'analyse de mutation est une solution pour cela ; le testeur crée des mutants, des versions erronées de la transformation sous test et évalue la capacité de ses oracles à détecter les fautes introduites dans les mutants. Cependant, l'analyse de mutation a plusieurs défauts : elle a un coût élevé, la création des mutants est une tâche manuelle et dépendante du langage de transformation utilisé, et l'exploitation des résultats pour améliorer les oracles est difficile.

Dans ce mémoire nous proposons de répondre à ces deux problématiques. Dans les chapitres qui suivent nous proposons une nouvelle fonction d'oracle qui produit un verdict partiel à propos d'une partie d'un modèle obtenu et une nouvelle approche pour évaluer la qualité d'un ensemble d'oracles.

CHAPITRE 3

Fonction d'oracle partielle pour le test de transformations de modèles

Dans cette thèse, nous étudions l'oracle du test de transformations de modèles. Nous souhaitons plus particulièrement assister le testeur dans sa tâche de création d'un oracle. Une première étape dans cette assistance est de lui fournir des outils adaptés à ses besoins.

Pour tester une transformation de modèles, le testeur doit écrire puis exécuter des cas de test. Un cas de test est principalement composé d'un modèle de test accompagné d'un oracle. Le but de l'oracle est de contrôler le modèle obtenu par l'exécution de la transformation sous test sur le modèle de test. Il doit s'assurer que ce modèle obtenu est correct vis-à-vis de la spécification de la transformation sous test. Un oracle est constitué d'une donnée d'oracle et d'une fonction d'oracle. La fonction d'oracle produit le verdict du test à partir de la donnée d'oracle, du modèle obtenu, et parfois du modèle d'entrée. Les fonctions d'oracles existantes [Mottu et al., 2008] utilisent principalement deux méthodes : (i) la comparaison du modèle obtenu avec un *modèle attendu* ou (ii) l'expression de contraintes pour vérifier des *propriétés attendues* sur le modèle de sortie.

Le testeur peut parfois vouloir ne contrôler qu'une partie d'un modèle obtenu. Dans certaines situations il est plus intéressant pour lui de ne prévoir la valeur attendue que d'une partie du modèle de sortie ; il crée ainsi un *modèle attendu partiel*. Nous distinguons trois cas de figure :

1. La transformation sous test manipule des modèles qui sont des données complexes, et le testeur ne peut pas maîtriser toute sa spécification. Dans ce cas, le testeur ne peut pas prévoir la totalité du modèle attendu pour un coût raisonnable.
2. La transformation sous test est endogène, elle modifie partiellement le modèle d'entrée (un ré-usinage par exemple) ; une partie du modèle reste ainsi inchangée. Dans ce cas, le modèle d'entrée peut être utilisé comme donnée d'oracle pour un test de non-régression. Le testeur peut ainsi s'assurer que la transformation sous test n'a pas d'effets de bord, qu'elle ne modifie pas des éléments qu'elle est censée ignorer.
3. La transformation sous test a des sorties polymorphes : comme pour nos cas d'étude, il existe plusieurs variantes valides du modèle de sortie. Ces variantes sont généralement sémantiquement équivalentes, mais syntaxiquement différentes. Cette variabilité vient de la spécification de la transformation de modèles ; le développeur fait un choix lors de la mise en œuvre. La création de l'oracle est faite en boîte noire, le testeur ne peut savoir quels choix ont été faits. Il doit prendre en compte toutes les variantes possibles.

Notre but est de contrôler une partie seulement d'un modèle de sortie en utilisant un modèle attendu partiel. Ce modèle attendu partiel correspond à la partie du modèle de sortie dont le testeur peut prédire la valeur avec un coût raisonnable. Comme nous l'avons vu dans la Section 2.3.2.4 (page 37), les fonctions d'oracle existantes ne correspondent pas à nos besoins. Pour la comparaison de modèles, le modèle attendu doit être entier. Les contraintes sur le modèle de sortie peuvent être exprimées sous la forme de contrats, mais écrire ces contrats est aussi compliqué qu'écrire la transformation sous test. D'autres fonctions d'oracles utilisent des assertions ou *patterns*, elles sont adaptées pour valider

individuellement des propriétés du modèle de sortie. Nous devons valider la partie du modèle obtenu qui nous intéresse comme un tout et non comme un ensemble de propriétés. Dans la suite de ce chapitre, nous nommons *partie contrôlée*, la partie du modèle obtenu à laquelle s'intéresse le testeur. Le reste du modèle obtenu constitue la *partie non-contrôlée*.

La contribution présentée dans ce chapitre est une nouvelle fonction d'oracle. À la différence des fonctions d'oracle utilisant la comparaison de modèles, cette nouvelle approche permet de ne contrôler qu'une partie d'un modèle obtenu. Cette approche utilise une comparaison de modèles partielle avec un modèle attendu pouvant lui aussi être partiel. Le modèle attendu est comparé au modèle obtenu, produit par la transformation sous test. Les différences observées entre ces deux modèles sont filtrées pour rejeter celles qui concernent la partie que le testeur ne souhaite pas contrôler. Pour créer ce filtre, nous devons identifier de manière précise la partie du modèle de sortie que nous ne souhaitons pas contrôler. Nous considérons qu'un élément du modèle de sortie appartient à la partie non contrôlée en fonction de l'élément du méta-modèle dont il est instance. Nous proposons de filtrer le résultat de la comparaison de modèles à l'aide d'un *pattern* extrait du méta-modèle de sortie de la transformation.

Pour valider cette approche, nous l'avons mise en œuvre dans un outil et nous avons utilisé cet outil sur nos cas d'étude aux sorties polymorphes. Les entrées de notre outil sont un modèle obtenu, un modèle attendu éventuellement partiel, ainsi que des *patterns* sous la forme de fragments de méta-modèle. Nous définissons 94 modèles attendus partiels avec 2 632 éléments, soit 70 % de moins qu'avec l'approche classique de comparaison avec un modèle attendu. Nous créons 94 modèles de test et obtenons 94 verdicts partiels avec nos oracles partiels, nous permettant de détecter 4 bugs.

Ce chapitre est organisé de la manière suivante : Dans la Section 3.1 nous détaillons notre approche pour définir un oracle partiel pour contrôler une partie d'un modèle de sortie. Dans la Section 3.2 nous présentons notre mise en œuvre de l'approche. Enfin nous détaillons nos expériences dans la Section 3.3 et nous discutons des résultats obtenus.

3.1 Proposition d'une fonction d'oracle partielle

Nous proposons une nouvelle fonction d'oracle partielle pour le test de transformations de modèles. Si le testeur souhaite ne contrôler qu'une partie d'un modèle obtenu, notre fonction d'oracle lui permet d'obtenir un verdict partiel concernant cette partie.

3.1.1 Donnée d'oracle partielle pour contrôler une partie d'un modèle de sortie

Notre objectif est de fournir au testeur une nouvelle fonction d'oracle, pour contrôler une partie d'un modèle obtenu. Les contrats génériques sont trop complexes, et la compa-

raison avec un modèle attendu a besoin de la totalité du modèle attendu. Nous proposons d'adapter la comparaison de modèles pour ne prendre en compte que la partie du modèle qui intéresse le testeur. Nous proposons de filtrer le résultat de la comparaison de modèles.

Dans la situation qui nous intéresse, le testeur souhaite ne contrôler qu'une partie du modèle obtenu. Le testeur peut, avec un coût raisonnable, créer un modèle attendu partiel qui correspond à la valeur attendue de la partie contrôlée du modèle obtenu. Ce modèle attendu partiel est conforme à une version relâchée du méta-modèle de sortie, comme Sen et al. le font pour le méta-modèle d'entrée [Sen et al., 2012]. Un méta-modèle relâché est un méta-modèle dont les contraintes ont été assouplies. Il n'y a aucun invariant et seules les références ont toutes des multiplicités de `0..1` ou alors `0..*`. Nous proposons d'utiliser ce modèle attendu partiel comme donnée d'oracle.

Nous proposons de comparer chaque modèle obtenu au modèle attendu partiel qui lui correspond. Nous filtrons ensuite le résultat de cette comparaison pour éliminer les différences observées à propos d'éléments de la partie non contrôlée. Pour ce filtrage, nous avons besoin d'une description de la partie non contrôlée afin d'identifier les différences observées à éliminer. Nous identifions les éléments de la partie non contrôlée d'après l'élément du méta-modèle de sortie (méta-classe, attribut ou référence) dont ils sont des instances. Nous fournissons des *patterns* au filtre. Ces *patterns*, sont des fragments du méta-modèle de sortie, ils sont définis une seule fois pour la transformation sous test. S'il peut le fournir, le testeur peut utiliser un modèle attendu entier à la place du modèle attendu partiel ; les différences observées qui concernent la partie non contrôlée de ce modèle attendu seront éliminées par le filtrage. Par exemple si la transformation sous test est un ré-usinage, il peut utiliser le modèle de test comme modèle attendu pour s'assurer que la transformation n'a pas d'effet de bord.

La Figure 3.1 illustre le déroulement de l'approche que nous proposons pour contrôler une partie d'un modèle de sortie obtenu. Pour chaque cas de test, nous proposons les étapes suivantes :

1. création du modèle attendu partiel et des patterns par le testeur ;
2. transformation du modèle de test par la transformation sous test ;
3. comparaison du modèle obtenu avec le modèle attendu qui lui correspond ;
4. filtrage du résultat de la comparaison à l'aide des *patterns* ;
5. observation du résultat filtré pour produire le verdict ;
   - s'il est vide, alors aucune différence n'est observée entre le modèle obtenu et le modèle attendu pour la partie contrôlée : le test passe ;
   - s'il n'est pas vide, alors il existe au moins une différence entre la partie contrôlée des deux modèles : le test échoue.

Dans notre exemple, la partie non contrôlée est composée des états finaux et des transitions qui les ciblent. Dans la Figure 2.7, les deux transitions ont la même cible, alors que dans la Figure 2.8 chacune a une cible différente. La Figure 3.2 présente les *patterns* décrivant la partie non contrôlée. Nous filtrons toutes les instances de FinalState et

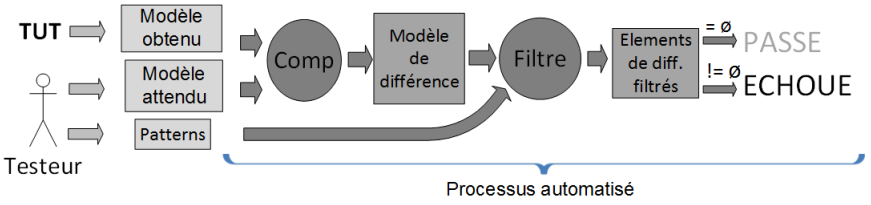
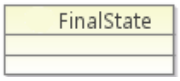
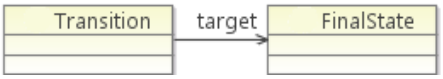


Figure 3.1 – Schéma de principe pour produire un verdict partiel



(a) Un état final



(b) Une transition ciblant un état final

Figure 3.2 – *Patterns* pour définir la partie non contrôlée pour la transformation de mise à plat d'une machine à état

celles de Transition qui ciblent une instance de FinalState. Dans nos expériences nous prenons aussi en compte les gardes (cf. Figure 3.3), actions, et évènements portés par les transitions (Section 3.3, page 60). Les *patterns* utilisés sont détaillés dans l'annexe A (page 105).

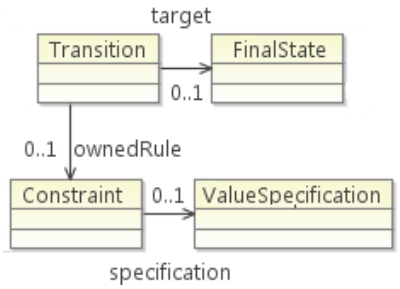


Figure 3.3 – Une garde portée par une transition qui cible un état final

Cette approche est automatisée au sein d'un outil. Cet outil prend en entrée un modèle obtenu, le modèle attendu (partiel ou non) qui lui correspond et les *patterns* de définition de la partie non contrôlée. Il fournit en sortie, le verdict partiel du test. Nous présentons cet outil dans la Section 3.2.

Nous pouvons également utiliser notre approche sans fournir de *patterns* pour vérifier que le modèle attendu partiel est inclus dans le modèle obtenu.

1. comparaison des deux modèles ;
2. filtrage : élimination des différences qui indiquent des ajouts dans le modèle obtenu ;
3. observation du résultat de la comparaison filtrée :
   – s'il est vide, alors le modèle attendu partiel est inclus dans le modèle obtenu : le test passe ;

– sinon, le modèle attendu partiel n'est pas inclus dans le modèle obtenu : le test échoue.

L'avantage de cette méthode est qu'elle est simple à mettre en place et à utiliser, le testeur fournit uniquement le modèle attendu partiel. L'inconvénient de cette méthode est son manque de précision ; en effet, la partie qui intéresse le testeur n'est pas contrôlée entièrement. Si la partie contrôlée du modèle obtenu contient des éléments en plus de ceux présents dans le modèle attendu partiel, le test passe quand même. Avec notre filtrage à base de *patterns*, nous conservons la différence observée sur la partie contrôlée des deux modèles de sortie : le test échoue.

3.1.2 Avantages de notre approche et complémentarité avec les autres fonctions d'oracle

Avec notre fonction d'oracle partielle, un seul modèle attendu, partiel ou non, suffit à détecter des fautes. L'approche classique par comparaison de modèles (cf. Section 2.3.2.4, page 37) n'en détecte aucune sans un modèle attendu entier. Les *patterns* utilisés pour le filtrage sont écrits une seule fois pour tous les cas de test d'une transformation donnée. De plus ils sont construits d'une manière familière pour un testeur familier de l'IDM, extraits d'un méta-modèle. Contrairement à un langage de comparaison dédié tel qu'ECL (Epsilon Comparison Language) [Kolovos et al., 2009], ou la spécialisation du moteur de comparaison pour chaque transformation, il n'a pas à apprendre un nouveau langage.

Cette nouvelle fonction d'oracle, ne remplace pas celles qui existent déjà, elle s'y ajoute. Chaque fonction d'oracle a ses avantages et ses inconvénients. La comparaison de modèles filtrée comble un défaut commun à celles déjà existantes : elles ne permettent pas de contrôler efficacement une partie seulement d'un modèle obtenu. Pour valider la transformation sous test il faut pouvoir entièrement contrôler le modèle obtenu. La comparaison de modèles filtrée ne fournit qu'un verdict partiel ; nous avons besoin de contrôler la partie du modèle obtenu qu'elle ignore.

Pour la partie non contrôlée, nous proposons d'utiliser des contrats partiels, comme les tracts de Vallecillo et al. [Vallecillo et al., 2012]. Ces contrats partiels sont plus simples car dédiés au contrôle d'une petite partie du modèle obtenu. Le risque d'avoir des fautes dans ces contrats partiels est donc faible. Dans le cas où la transformation sous test effectue un ré-usinage, nous proposons d'utiliser la comparaison de modèles partielle pour contrôler la partie du modèle non modifiée par la transformation, et des contrats partiels pour la partie modifiée. En écrivant les contrats le testeur peut se concentrer sur le fonctionnement attendu de la transformation sans s'occuper du reste.

3.2 Mise en œuvre de l'approche proposée

Pour pouvoir valider notre approche, nous l'avons mise en œuvre dans un outil. Nous décrivons cette mise en œuvre ici. Après une description de l'environnement technique dans lequel nous nous sommes placés, nous détaillons le processus de traitement des *patterns* utilisés pour le filtrage.

3.2.1 Environnement technique

Sur un plan technique, notre fonction d'oracle permet de tester des transformations produisant des modèles au format XMI [1], comme dans l'Eclipse Modeling Framework [2]. Nous avons choisi EMF pour son utilisation répandue dans le monde académique et les nombreux outils disponibles.

Phase de comparaison Pour effectuer la comparaison de modèles, nous utilisons EMFCompare [3] version 1.2. EMFCompare est conforme aux critères exposés par Cicchetti [Cicchetti et al., 2007] pour des outils de comparaison de modèles. Pour chaque comparaison, EMFCompare produit deux modèles : le *Match model*, ou modèle de points communs, contient les éléments communs aux deux modèles, tandis que le *Diff model* ou modèle de différences, regroupe les différences observées. Dans le modèle de différences, une différence observée est définie comme une instance de la méta-classe `DiffElement`, comme le montre l'extrait du méta-modèle de différences d'EMFCompare présenté dans la Figure 3.4. Une différence observée peut concerner une méta-classe ou l'une de ses propriétés (un attribut ou une référence).

Phase de filtrage Pour filtrer le résultat de la comparaison des modèles (Figure 3.1), nous effectuons du *pattern matching* sur le modèle de différences produit par EMFCompare. Nos *patterns* décrivent une différence observée à propos d'un élément de la partie contrôlée des modèles comparés. Le test passe si l'application du *pattern* sur le modèle de différences retourne un résultat vide. Cela signifie que rien dans le modèle de différences ne correspond à nos *patterns*, donc aucune différence n'est observée à propos de la partie contrôlée.

Le testeur écrit des *patterns* comme des fragments de méta-modèle au format Ecore. Chaque fragment décrit un élément du méta-modèle qui appartient à la partie non contrôlée du modèle (un état final par exemple dans la Figure 3.2(a)). Nous ne souhaitons pas forcément filtrer toutes les différences observées à propos d'instances d'un élément donné

1. http ://www.omg.org/spec/XMI/
2. http ://www.eclipse.org/emf/
3. http ://www.eclipse.org/emf/compare/

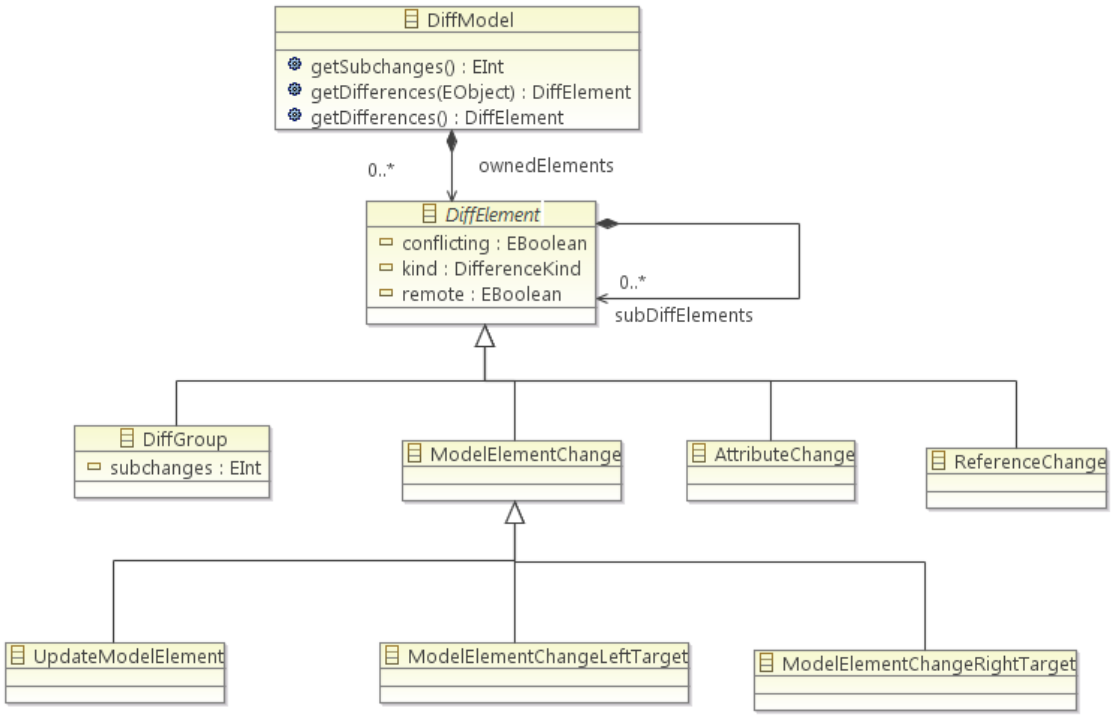


Figure 3.4 – Extrait du méta-modèle de différence d'EMFCompare

du méta-modèle. Nous pouvons préciser notre définition en ajoutant dans les fragments, d'autres méta-éléments. Le *pattern* de la Figure 3.2(b) permet, par exemple, de filtrer les instances de `Transition` dont la cible est une instance de `FinalState`. Nous utilisons des fragments de méta-modèles composés de méta-classes et de leurs propriétés (attributs et références). Chaque fragment contient au plus un attribut, celui que nous voulons filtrer. Ainsi nous pouvons filtrer n'importe quel élément à propos duquel EMFCompare peut observer une différence.

Le filtrage est effectué par *pattern matching*. Pour cela nous utilisons l'outil EMF IncQuery [4]. Avec IncQuery les *patterns* sont écrits dans un format textuel. Nous devons donc transformer les fragments de méta-modèle fournis par le testeur en *patterns* IncQuery. Nous avons développé cette transformation en Java. Pour cela, nous transformons automatiquement les fragments de méta-modèle en *patterns* IncQuery, à l'aide d'une transformation écrite en Java. Cette transformation est intégrée au filtre (Figure 3.1).

3.2.2 Traitement automatique des *patterns*

Dans cette Section, nous détaillons le fonctionnement de la transformation de génération des règles IncQuery à partir des fragments de méta-modèle fournis par le testeur.

4. http ://incquery.net/

Cette transformation prend en entrée :
- les fragments de méta-modèle qui définissent la partie non contrôlée du modèle de sortie ;
- pour chaque fragment de méta-modèle, le nom de l'élément que le testeur souhaite filter ;
- le nom de la méta-classe racine de chaque fragment.

La transformation commence à parcourir le fragment à partir de la méta-classe racine. Par exemple, dans le fragment de la Figure 3.2(b), `Transition` est l'élément que le testeur souhaite filtrer ainsi que la méta-classe racine. La transformation produit des règles IncQuery qui sont applicables sur le résultat de la comparaison pour retourner les différences concernant uniquement la partie contrôlée.

Cette transformation se déroule en trois étapes successives.

Première étape Nous recherchons les différences entre deux modèles, donc nous devons définir comment une différence est représentée dans le modèle de différences d'EMFCompare. Nous filtrons le résultat d'une comparaison effectuée avec EMFCompare. Nous avons donc besoin d'un premier *pattern* pour définir une différence observée C'est le rôle du *pattern isDifference*. Il correspond à un élément quelconque pour lequel une différence est observée (un élément référencé dans une instance de `DiffElement`). Ce *pattern* générique et indépendant de la transformation sous test est utilisé par ceux générés à partir des fragments de méta-modèle. Il est présenté dans le Listing 3.1.

```
pattern isDifference(E) = {
        ModelElementChangeLeftTarget(ME);
        ModelElementChangeLeftTarget.leftElement(ME, E);
}or{
        ModelElementChangeRightTarget(ME);
        ModelElementChangeRightTarget.rightElement(ME, E);
}or{
        UpdateModelElement(UME);
        UpdateModelElement.leftElement(UME, E);
}or{
        UpdateModelElement(UME);
        UpdateModelElement.rightElement(UME, E);
}or{
        AttributeChange(AC);
        AttributeChange.leftElement(AC, E);
}or{
        AttributeChange(AC);
        AttributeChange.rightElement(AC, E);
}or{
        ReferenceChange(RC);
        ReferenceChange.leftElement(RC, E);
}or{
        ReferenceChange(RC);
        ReferenceChange.rightElement(RC, E);
}
```

Listing 3.1 – Une différence observée à propos d'un élément

Deuxième étape Dans cette deuxième étape, nous générons les *patterns* Incquery correspondant aux fragments définis par le testeur, à chaque fragment correspond un *pattern*. Par exemple, les fragments de la Figure 3.2 sont transformés pour produire les *patterns* Inquery présentés dans le Listing 3.2. Pour chaque fragment, nous créons l'entête du *pattern* avec son nom (nous reprenons celui du fragment) et son paramètre dans lequel les éléments identifiés sont collectés.

Chaque méta-classe est transformée en une règle pour identifier les éléments du modèle qui en sont instances. Par exemple la méta-classe `FinalState` dans le fragment de méta-modèle de la Figure 3.2(a) est transformé en la règle `FinalState(F)` pour collecter dans la variable `F` les instances de `FinalState` présentes dans le modèle. Lorsque les *patterns* expriment plus de contraintes, la transformation génère des règles supplémentaires. Par exemple, le *pattern* de la Figure 3.2(b) devient la règle `finalTransition(T)` : (i) les méta-classes `Transition` et `FinalState` deviennent les règles `Transition(T)` et `FinalState(F)` pour collecter dans les variables T et F, les instances de `Transition` et `FinalState` présentes dans le modèle, (ii) de plus, la référence `target` entre `Transition` et `FinalState` devient la règle `Transition.target(T, F)` pour préciser que nous souhaitons obtenir dans `T` les transitions dont la cible est un état final.

```
// A FinalState F
pattern finalState(F) = {
        FinalState(F);
}

// A transition targeting a final State
pattern finalTransition(T) = {
        Transition(T);
        FinalState(F);
        Transition.target(T, F);
}
```

Listing 3.2 – La partie non contrôlée exprimée en Incquery

Troisième étape Dans la troisième et dernière étape, nous générons les règles qui fournissent le verdict de la comparaison filtrée. Dans la première étape, nous avons décrit un élément à propos duquel une différence est observée. Dans la deuxième étape, nous avons défini les éléments de la partie non contrôlée. Dans cette dernière étape, nous recherchons tous les éléments de la partie non contrôlée à propos desquels une différence est observée. C'est pourquoi, nous combinons les règles des étapes précédentes pour en former une nouvelle.

Dans le Listing 3.3 nous recherchons tout élément pour lequel une différence est observée, mais qui n'est pas une instance de `FinalState` ou de `Transition` ciblant une instance de `FinalState`. Si ce *pattern* ne trouve rien, alors aucune différence n'est observée dans la partie contrôlée entre les modèles de sortie et attendus : le test passe.

```
/* A is a Difference which is not about a FinalSate
  or a Transition targeting one FinalState
  If the result is empty then the test pass for the common part */
pattern verdictPassIfAEmpty(A) = {
        find isDifference(A);
        neg find finalState(A);
        neg find finalTransition(A);
}
```

Listing 3.3 – *Pattern* pour le verdict de la partie contrôlée

3.3 Validation expérimentale

Maintenant que nous disposons d'une mise en œuvre de notre approche, nous pouvons valider cette dernière de manière expérimentale. Nous avons écrit des oracles partiels pour nos deux cas d'étude, les transformations de modèles aux sorties polymorphes présentées dans la Section 1.2 (page 6). Pour la transformation de mise à plat de la machine à états, nous utilisons pour nos expériences la mise en œuvre qui transforme des modèles conformes au méta-modèle UML non simplifié (Section 1.2.1.2, page 7). Le matériel expérimental utilisé est disponible en ligne [Finot et al.,].

3.3.1 Protocole d'expérimentation

L'objectif des expériences menées est de répondre à trois questions :
- **Question 1 :** Un testeur peut-il écrire des modèles attendus partiels et des *patterns* ?
- **Question 2 :** Ces *patterns* et modèles attendus partiels peuvent-ils être traités par une fonction d'oracle partielle pour produire un verdict partiel ?
- **Question 3 :** L'approche proposée est-elle plus adaptée que les autres fonctions d'oracle pour contrôler uniquement une partie d'un modèle obtenu, et ainsi produire un verdict partiel ?

Pour répondre à la première question, nous créons des modèles de test pour nos deux cas d'étude. Puis, nous créons les données d'oracles correspondantes : les modèles attendus et les *patterns*. Nous utilisons des modèles attendus partiels qui correspondent uniquement à la partie contrôlée. La partie contrôlée est la partie commune à toutes les variantes du modèle de sortie.

La première tâche à effectuer lors de la définition de cas de test est la sélection des données de test. Notre étude ne concerne pas les données de test, nous utilisons une approche existante pour la génération de données de test.

Nous avons choisi d'appliquer l'approche proposée par Fleurey et al. [Fleurey et al., 2009]. Pour chaque cas d'étude, à partir du méta-modèle d'entrée nous avons créé des fragments

de modèles en choisissant l'une des stratégies proposées, puis nous avons écrit des modèles de test à partir de ces fragments.

Pour la première version de la transformation de mise à plat de la machine à états (celle utilisant le méta-modèle UML) complet, nous avons créé dix fragments de modèles en utilisant la stratégie IFClassΣ. Avec cette stratégie, un seul fragment de modèle est créé par méta-classe, et chaque valeur représentative des propriétés de cette méta-classe est présente au moins une fois. Avec cette stratégie, nous ne créons que peu de fragments, mais chacun de ces fragments contient un nombre plus important d'éléments qu'avec une autre stratégie. A partir de ces dix fragments, nous créons trente modèles de test, chaque fragment donne trois modèles selon les critères suivants :

- un modèle avec cinq états simples, dix transitions et aucun composite ;
- un modèle avec cinq états dont deux composites et dix transitions ;
- un modèle avec dix états dont cinq composites et quinze transitions.

Pour notre second cas d'étude, nous créons huit fragments de modèles en appliquant la stratégie IFCombΠ. Avec cette stratégie pour chaque méta-classe, toutes les combinaisons possibles des propriétés de cette méta-classe doivent être présentes dans les fragments ; un fragment correspond à une combinaison. Ce nombre limité de fragments est dû à la structure du méta-modèle de sortie de la transformation. Dans ce méta-modèle, il y a beaucoup de liens d'héritage, peu de références, et presque aucun attribut. Nous avons choisi encore une fois de créer plusieurs modèles correspondant à chaque fragment de modèle. Pour couvrir chaque méta-classe concrète, si un fragment contient une référence vers une méta-classe mère nous créons un modèle pour chacune des méta-classes filles non abstraites. Après avoir éliminé toutes les combinaisons incorrectes, nous créons 64 modèles de test.

Pour répondre à la deuxième question, nous transformons nos modèles de tests et nous utilisons nos oracles partiels pour contrôler les modèles obtenus et produire des verdicts.

Pour répondre à la troisième question, nous comparons la taille des modèles attendus partiels à celle des modèles entiers nécessaires sans notre proposition.

3.3.2 Résultats obtenus

Les résultats obtenus sont présentés dans les Tables 3.1 et 3.2. Pour chaque modèle attendu partiel, nous présentons le nombre de ses éléments, le nombre d'éléments dans le modèle attendu entier, la proportion de la partie contrôlée dans le modèle, ainsi que le nombre de variantes existantes et le verdict partiel du test. Nous présentons les résultats obtenus pour chacun de nos cas d'étude, puis nous en discutons pour répondre aux questions posées.

modèle de référence	# éléments	# éléments de la partie contrôlée	% partie contrôlée	#variantes	verdict partiel
1	36	32	89 %	1	pass
2	40	33	83 %	1	pass
3	37	30	81 %	2	pass
4	42	35	83 %	2	pass
5	35	31	89 %	1	pass
6	37	29	78 %	2	pass
7	42	32	76 %	5	pass
8	39	32	82 %	2	pass
9	42	29	69 %	93	fail
10	29	22	76 %	2	pass
11	22	17	77 %	2	pass
12	25	18	72 %	2	pass
13	23	18	78 %	2	pass
14	17	12	71 %	2	fail
15	18	13	72 %	2	pass
16	20	14	70 %	2	pass
17	25	18	72 %	5	pass
18	19	14	74 %	2	pass
19	18	12	67 %	5	pass
20	18	11	61 %	5	pass
21-30 moy :	15	13	87 %	1	pass
moyenne	35	28	79 %	6.71	
somme	734	582		157	

Table 3.1 – Résultats observés pour la transformation de mise à plat d'une machine à état

Mise à plat d'une machine à états UML Parmi les 30 modèles de test créés, 10 ne contiennent aucun état composite à transformer. Le modèle attendu pour leur transformation sera donc identique au modèle de test. Pour ce qui est des modèles restants, le modèle attendu est la partie contrôlée commune à toutes les variantes.

4 oracles partiels indiquent un échec du test. Dans deux des modèles il manquait une transition. Les deux autres échecs s'expliquent par une garde manquante sur une transition nouvellement créée ; plus précisément, la garde a été créée mais n'a pas été associée à la transition.

Pour la majorité des cas de test, le nombre de variantes est limité (1, 2, ou 5), mais l'un des modèles de sortie possède 93 variantes (modèle 9 dans la table 3.1). Ce modèle contient 42 éléments dans sa globalité et 29 éléments dans sa partie contrôlée. Dans ce cas précis, avec l'approche classique de comparaison de modèles, le testeur devrait créer 93 modèles attendus, environ 3,906 éléments (42 * 93), et tous les comparer au modèle obtenu, pour produire un verdict.

Néanmoins, la majorité des éléments du modèle de sortie appartiennent à la partie contrôlée. Le testeur pourrait créer un seul modèle avec la partie contrôlée, puis copier ce modèle autant de fois que nécessaire pour ajouter ensuite dans chacun de ces modèles la partie non contrôlée. Cependant, pour le cas de test avec 93 variantes, il restera au testeur la création manuelle de 1 238 éléments (29 + (42 - 29) * 93) ainsi que l'exécution

modèle de attendu	# éléments	# éléments de la partie contrôlée	% partie contrôlée	#variantes	verdict partiel
1	38	32	84 %	1	pass
2	3	3	100 %	1	pass
3	38	32	84 %	1	pass
4	38	32	84 %	2	pass
5	38	35	92 %	2	pass
6	38	30	79 %	2	pass
7	42	36	86 %	4	fail
9	48	40	83 %	4	pass
8	42	36	86 %	4	fail
10	42	35	83 %	6	pass
11	38	31	82 %	6	pass
12	38	31	82 %	6	pass
13	34	29	85 %	12	pass
14	44	39	89 %	12	pass
15	44	36	82 %	12	pass
16	48	40	83 %	24	fail
17	48	40	83 %	24	fail
18	50	43	86 %	96	pass
19	50	43	86 %	96	pass
20	50	43	86 %	96	pass
21-64 moy :	36	31	88 %	6	pass
moyenne	37.3	32.03	87 %	10.6	
somme	2388	2050		678	

Table 3.2 – Résultats observés pour la transformation UML vers CSP

des 93 comparaisons de modèles avant d'obtenir le moindre verdict. La transformation occupe 500 lignes de code, nous avons écrit 510 lignes de contrats pour détecter les mêmes erreurs qu'avec notre approche. Les contrats sont plus complexes que la transformation sous test et ne couvrent pourtant pas entièrement la partie contrôlée des modèles de sortie. Ils ne contrôlent par exemple pas les effets des transitions. Notre approche au contraire permet de contrôler la partie contrôlée dans sa totalité, elle est présente dans nos modèles attendus partiels. Nous avons écrit 153 lignes de contrats pour la partie non commune, non contrôlée par notre approche.

Diagramme d'activités vers CSP La transformation sous test est exogène, donc chaque modèle obtenu est différent du modèle de test. Parmi les 64 modèles de test, il n'existe qu'une seule variante pour 7 d'entre eux. 4 tests échouent, lorsque au moins 2 instances de `JoinNode` sont présentes dans le modèle d'entrée, seule la première est correctement transformée.

Le nombre maximal de variantes est de 96 pour 3 modèles. Sans notre approche, le testeur devrait créer les 96 modèles attendus : 4 800 éléments (502 * 96) ou 715 éléments (43 + (50 - 43) * 96) en n'écrivant qu'une seule fois la partie contrôlée.

Nous écrivons encore une fois des contrats aussi complexes que la transformation sous test (218 lignes de contrats contre 210 pour la transformation). Les contrats

détectent les mêmes fautes que nos oracles partiels, mais ne contrôlent pas entièrement la partie commune à toutes les variantes. Par exemple, l'ordre des instances de `ProcessAssignment` est contrôlé par notre fonction d'oracle partielle, mais pas par les contrats. Nous écrivons 100 lignes de contrats pour la partie non commune, non contrôlée par notre approche.

Discussion A partir des résultats présentés ci-dessus nous répondons aux questions posées dans le protocole d'expérimentation (Section 3.3.1).

Pour répondre à la **Question 1**, nous avons créé des modèles attendus partiels et des *patterns*. Les modèles attendus partiels correspondent à la partie des modèles de sortie qui nous intéresse.

Les *patterns* sont des fragments de méta-modèle qui décrivent la partie des modèles de sortie qui ne nous intéresse pas. Nous écrivons des modèles attendus partiels et des *patterns* pour filtrer la partie non contrôlée des modèles de sortie. Cette transformation a des sorties polymorphes, de ce fait nous pouvons créer des modèles attendus entiers, mais certains ne sont qu'une variante parmi plusieurs possibilités.

Pour répondre à la **Question 2**, nous utilisons nos modèles attendus partiels et *patterns* en tant que données d'oracle.

Pour répondre à la **Question 3**, nous obtenons des verdicts partiels avec 94 modèles attendus partiels (2 632 éléments) accompagnés de 8 *patterns* (18 éléments), et 94 comparaisons de modèles. Avec la fonction d'oracle classique de comparaison avec un modèle attendu, nous aurions eu besoin de 835 modèles (36 184 éléments ou 8 677 éléments en copiant les parties contrôlées) et 835 comparaisons de modèles. Nous avons donc un gain de 93 % en matière d'éléments de modèle (70 % si copie de la partie contrôlée) nous réduisons de 89 % le nombre de comparaisons de modèles. Nos cas d'étude manipulent des modèles de petite taille, avec un nombre moyen d'éléments de 35 et 37 respectivement pour chaque cas d'étude. Manipuler des modèles de plus grande taille entrainerait un gain plus important. Ce gain serait décisif pour le testeur, s'il devait écrire manuellement les variantes des modèles attendus.

Nous écrivons des contrats qui détectent les même fautes que nos oracles partiels dans la partie contrôlée de nos modèles obtenus. Ces contrats ne contrôlent pas entièrement la partie contrôlée par nos oracles partiels. Le risque de trouver des fautes est aussi élevé pour les contrats que pour la transformation sous test. L'utilisation de contrats génériques n'est donc pas adaptée à nos besoins.

Nous ne contrôlons pas entièrement la correction du modèle de sortie. Cependant, nos oracles partiels détectent des fautes dans nos deux transformations, pour l'une d'entre elles la mise en œuvre n'a pas été réalisée par nous [Holt et al., 2009]. De plus, la partie contrôlée est une partie significative de nos modèles de sortie (plus de 61 % ici). Enfin, les transformations n'agissent pas uniquement sur la partie non contrôlée des modèles. Dans

les deux cas d'étude, la partie contrôlée est modifiée par la transformation, elle n'est pas présente telle quelle dans le modèle d'entrée. Dans la première transformation, les états simples ne sont pas modifiés, mais les transitions qui ciblent un état composite ou en sortent le sont (entre A et B dans les Figures 2.7, page 27 et 2.8, page 28). La seconde transformation est exogène, d'un langage vers un autre, les méta-modèles d'entrée et de sortie sont différents. Ce verdict partiel est une bonne information pour le testeur.

Nous pouvons donc conclure que, dans le cas où le testeur souhaite contrôler une partie seulement d'un modèle obtenu, notre fonction d'oracle partielle est la plus adaptée.

Nous proposons d'utiliser des contrats partiels pour compléter notre verdict partiel. Nous avons montré que des contrats génériques ne sont pas adaptés pour contrôler entièrement un modèle de sortie. Une partie des contrats que nous écrivons pour nos expériences contrôle la partie des modèles obtenus non contrôlée par nos oracles partiels. En ne conservant que cette partie des contrats, nous obtenons des contrats plus petits et donc plus simples (nous réduisons leur taille de 69,4 % pour le premier cas d'étude et 52,4 % pour le second). Le risque de trouver une faute dans ces contrats est donc plus faible. L'utilisation de contrats partiels est donc une bonne solution pour compléter le verdict partiel obtenu avec notre fonction d'oracle.

3.3.3 Validité des expériences : analyse critique

Nous avons appliqué notre approche avec succès pour construire des oracles partiels pour le test de deux transformations de modèles. La principale question sur la validité de nos expériences est la question de la représentativité de nos cas d'étude. Ces deux cas d'étude, ne représentent pas la totalité des transformations de modèles possibles, mais en donnent un bon aperçu. En effet, le premier cas d'étude est un ré-usinage, il est endogène et modifie uniquement une partie du modèle d'entrée. Cette transformation traite des modèles UML, les modèles d'entrée et de sortie sont conformes au méta-modèle UML. Le second cas d'étude est une transformation exogène, les méta-modèles d'entrée et de sortie sont différents. Ces transformations sont mises en œuvre dans des langages différents, l'une est en ATL, et l'autre en Kermeta. De plus notre fonction d'oracle n'est pas dépendante du langage de transformation utilisé.

L'identification de la partie non contrôlée des modèles obtenus est une étape nécessaire pour l'utilisation de notre approche. Cette tâche reste manuelle, comme d'autres tâches du test logiciel. Pour identifier de manière précise la partie non contrôlée, le testeur doit maîtriser la spécification de la transformation sous test. C'est de cette spécification qu'il extrait les *patterns* utilisés pour le filtrage. Nous n'avons pas défini de méthode pour effectuer cette tâche de manière systématique, cela reste une perspective de ce travail. Cependant la présence de certains éléments dans une spécification peut suggérer que la transformation a des sorties polymorphes. C'est par exemple le cas lorsque le modèle de sortie contient des opérateurs binaires, comme les programmes CSP de notre second

cas d'étude. Une étude approfondie de ces éléments devrait permettre de confirmer la présence d'un polymorphisme et d'identifier précisément la partie non contrôlée. Lorsque la transformation sous test est un ré-usinage, la partie du méta-modèle présente dans la spécification est celle qui sera modifiée. Cette partie est donc la partie non contrôlée.

Enfin on peut aussi s'interroger sur l'expressivité des *patterns* utilisés pour le filtrage. Ces *patterns* ne permettent pas de filtrer selon une valeur spécifique d'une propriété, par exemple il n'est pas possible de filtrer uniquement les différences portant sur des états dont le nom est "A". Cette impossibilité est due à la définition de nos *patterns* sous la forme de fragments de méta-modèle. Nous avons choisi d'utiliser des fragments de méta-modèle pour être indépendants vis-à-vis des cas de test. Nos *patterns* sont définis une seule fois pour une transformation sous test donnée et sont valides pour tous les cas de test. Nos *patterns* permettent d'ajouter des contraintes sur les éléments à filtrer ; pour notre premier cas d'étude nous avons par exemple filtré les différences qui concernent des transitions dont la cible est un état final. Nos *patterns* sont adaptés pour les cas d'étude sur lesquels nous avons expérimenté mais il serait intéressant de les améliorer pour pouvoir prendre en compte d'autres situations.

3.4 Conclusion

Dans ce chapitre, nous nous sommes intéressés à la question **A** posée dans la Section 1.1 (page 2). Pour tester une transformation de modèles le testeur dispose de plusieurs fonctions d'oracles ; mais elles ne sont pas toujours adaptées à tous les cas de figure. C'est par exemple le cas, si le testeur ne souhaite contrôler qu'une partie d'un modèle obtenu, cela est impossible avec une comparaison de modèles, et difficile avec des contrats génériques.

Dans ce chapitre, nous avons proposé une nouvelle fonction d'oracle partielle. Cette fonction d'oracle permet de contrôler une partie d'un modèle produit par la transformation sous test. Nous comparons le modèle obtenu avec un modèle attendu, partiel ou non. Nous filtrons ensuite le résultat de cette comparaison ; nous éliminons toutes les différences observées à propos d'éléments qui n'appartiennent pas à la partie contrôlée.

Nous avons implanté notre approche dans un outil pour la valider expérimentalement. Ces expériences ont porté sur les deux cas d'étude présentés dans la Section 1.2 (page 6), pour contrôler la partie commune aux différentes variantes des modèles obtenus. Notre approche, moins coûteuse que les approches classiques, permet de détecter des erreurs dans les modèles obtenus, et donc des fautes dans la transformation.

Cette nouvelle fonction d'oracle ne remplace pas celles déjà existantes. Elle s'y ajoute pour combler un manque que nous avons identifié. Chaque fonction d'oracle peut être plus adaptée qu'une autre suivant la situation où le testeur se trouve. Ainsi, nous proposons

d'utiliser des contrats pour valider la partie non contrôlée par notre nouvelle fonction d'oracle partielle.

Dans ce chapitre nous avons proposé une assistance passive au testeur. Nous avons fourni une nouvelle fonction d'oracle pour combler un manque de celles déjà existantes. Dans une prochaine étape, nous pouvons lui fournir une assistance plus active. Nous allons nous intéresser à la qualité des oracles créés à l'aide des fonctions d'oracle disponibles.

CHAPITRE 4

Qualification d'oracles pour le test de transformations de modèles

Nous poursuivons l'objectif de l'assistance au testeur dans la création d'oracles pour le test de transformations de modèles. Dans le chapitre précédent, nous nous sommes intéressés aux outils à sa disposition. Nous avons identifié puis comblé un manque dans les fonctions d'oracle existantes. Nous avons proposé une nouvelle fonction d'oracle [Finot et al., 2012a, Finot et al., 2013a] pour contrôler une partie d'un modèle obtenu. Dans ce chapitre, nous assistons le testeur en évaluant la qualité des oracles d'une suite de tests. Une suite de tests consiste en un ensemble de cas de test. Chaque cas de test est constitué principalement d'un modèle d'entrée de la transformation sous test et d'un oracle. L'oracle est chargé de contrôler que le modèle obtenu en transformant le modèle de test est correct par rapport à la spécification de la transformation sous test.

La création d'une suite de tests pour une transformation de modèles se déroule en plusieurs étapes. Tout d'abord, le testeur sélectionne puis qualifie les modèles de test. Ensuite il crée les oracles qui correspondent aux modèles de test, puis il les qualifie. Nous avons présenté dans la section 2.3.3 (page 39) les approches existantes pour évaluer la qualité d'un ensemble de cas de test de transformations de modèles. L'évaluation de la qualité des modèles de test a déjà été étudiée [Fleurey et al., 2009], mais l'évaluation de la qualité d'un ensemble d'oracles a fait l'objet de peu d'études. L'analyse de mutation [DeMillo et al., 1978] est utilisée pour qualifier les oracles [Mottu et al., 2006b]. Elle évalue leur capacité à détecter des fautes réelles dans la mise en œuvre d'un système. Le testeur injecte volontairement des fautes dans la transformation sous test pour créer des mutants. Puis il transforme les modèles de test avec les mutants. Ensuite le testeur contrôle les modèles produits par ces mutants à l'aide des oracles. Un mutant est tué si une erreur est détectée dans au moins un des modèles qu'il a produits. Cependant, l'analyse de mutation a plusieurs limitations. Sa première limitation est qu'elle n'est pas totalement automatisée : si l'application des opérateurs de mutation peut être réalisée automatiquement, ce n'est pas le cas de la détection de mutants équivalents. La deuxième limitation de l'analyse de mutation est qu'elle est dépendante du langage de transformation utilisé : les opérateurs de mutation sont définis indépendants du langage utilisé, mais leur mise en œuvre varie. Sa troisième limitation est le temps nécessaire pour créer les mutants puis les exécuter sur tous les modèles de test. La quatrième et dernière limitation concerne l'amélioration des oracles, le testeur peut difficilement exploiter les résultats de l'analyse de mutation pour améliorer ses oracles.

Dans ce chapitre nous proposons une alternative à l'analyse de mutation pour évaluer la qualité de l'ensemble des oracles d'une suite de tests de transformations de modèles. Nous considérons que la qualité d'un ensemble d'oracles est sa capacité à détecter des fautes dans la transformation sous test. Nous proposons de mesurer la couverture du méta-modèle de sortie par cet ensemble d'oracles. Meilleure est la couverture du méta-modèle de sortie par les oracles et meilleurs sont ces oracles. Cette approche n'a pas les défauts de l'analyse de mutation. Elle est indépendante du langage de transformation utilisé, automatisable et moins coûteuse. La mesure de couverture du méta-modèle de sortie

fournit deux résultats : le taux de couverture du méta-modèle, et la liste des éléments non couverts. À partir de ce cette liste d'éléments, le testeur peut améliorer la couverture du méta-modèle. Il améliore ainsi la qualité de ses oracles.

Notre approche a été mise en œuvre dans un outil. Cet outil qualifie un ensemble d'oracles en mesurant leur couverture du méta-modèle de sortie. Pour valider notre proposition, nous comparons les résultats obtenus par notre outil avec ceux de l'analyse de mutation. Nous définissons des cas de test avec plusieurs ensembles d'oracles. Ces expériences démontrent que les ensembles d'oracles ayant une meilleure couverture tuent plus de mutants et détectent donc plus de fautes.

Ce chapitre est organisé de la manière suivante : dans la section 4.1 nous détaillons notre approche pour évaluer la qualité des oracles d'une suite de tests de transformations de modèles en nous basant sur leur couverture du méta-modèle de sortie. Dans la section 4.2 nous détaillons la mise en œuvre de notre approche. Dans la section 4.3 nous présentons les expériences que nous avons menées sur nos deux cas d'étude et discutons des résultats. Dans la section 4.4 nous présentons nos idées pour améliorer des oracles qualifiés avec notre approche.

4.1 Couverture du méta-modèle de sortie et qualité des oracles

Notre but est de fournir au testeur une alternative à l'analyse de mutation pour qualifier les oracles d'une suite de tests de transformations de modèles. Notre approche est indépendante du langage de transformation et automatisable. Elle est aussi moins coûteuse que l'analyse de mutation et fournit des indices pour améliorer les oracles. Pour qualifier un ensemble d'oracles, nous mesurons sa couverture de la spécification de la transformation sous test. Après avoir présenté notre approche, nous détaillons notre mesure de la couverture d'un méta-modèle par un modèle.

4.1.1 Proposition : qualification des oracles pour le test de transformations de modèles

Une suite de tests est un ensemble de cas de test. Un cas de test, lui, se compose principalement d'un modèle de test accompagné d'un oracle. L'oracle est chargé de contrôler que le modèle obtenu par la transformation du modèle de test est correct par rapport à la spécification de la transformation sous test.

Dans ce chapitre nous nous intéressons à la qualité des cas de test. Notre but est de répondre à la question **B** (Section 1.1, page 4), nous cherchons à évaluer la qualité des oracles d'une suite de tests.

Comme nous l'avons vu, une mesure de couverture est souvent utilisée pour évaluer la qualité de données de test (Section 2.1.4.1, page 2.1.4.1). Fleurey et al. [Fleurey et al., 2009] par exemple, mesurent la couverture du méta-modèle d'entrée par des modèles de test pour évaluer leur qualité. Nous proposons une approche similaire pour les oracles.

Comme l'analyse de mutation, notre approche évalue la qualité de l'ensemble des oracles d'une suite de tests. Pour nous, la qualité des oracles d'une suite de tests est leur capacité à détecter des fautes. Des oracles de meilleure qualité détecteront plus de fautes dans la transformation sous test. Selon les critères de Staats et al. [Staats et al., 2011], nous évaluons la puissance de ces oracles.

Notre évaluation de la qualité des oracles d'une suite de tests utilise une mesure de couverture. Nous mesurons la couverture du méta-modèle de sortie effectif de la transformation par les oracles. Si certains éléments du méta-modèle de sortie ne sont pas couverts par les oracles, alors ces éléments ne sont pas contrôlés par ces oracles. Dans le modèle obtenu, les oracles ne pourront pas détecter d'erreurs à propos d'instances de ces éléments. Ainsi, un ensemble d'oracles qui a une meilleure couverture du méta-modèle de sortie est de meilleure qualité.

Notre approche est présentée dans la Figure 4.1. Tant que le méta-modèle de sortie n'est pas entièrement couvert :

1. mesurer la couverture du méta-modèle de sortie de la transformation par les oracles ;
2. évaluer le taux courant ;
3. effectuer un diagnostic et proposer des améliorations (détaillé dans la Section 4.4).

Fin tant que.

Il n'est pas toujours possible d'obtenir 100% de couverture. La transformation sous test ne manipule pas forcément tous les éléments du méta-modèle de sortie. Pour cette raison, nous proposons d'utiliser le méta-modèle de sortie effectif de la transformation. Le testeur ne conserve que les éléments du méta-modèle de sortie qui sont manipulés par la transformation sous test, d'après sa spécification. Nous mesurons la couverture de ce méta-modèle de sortie effectif par les oracles de la suite de tests.

La mesure de couverture fournit deux résultats : le taux de couverture du méta-modèle de sortie effectif par les oracles et la liste des éléments non couverts. Le but du testeur est de maximiser la couverture pour obtenir un ensemble d'oracles de qualité. Un taux de couverture inférieur à 100 %, signifie que les oracles sont améliorables. Pour améliorer la qualité des oracles, le testeur va donc tenter d'améliorer la couverture du méta-modèle de sortie effectif. Pour cela, il peut utiliser la liste des éléments du méta-modèle de sortie effectif non couverts par les oracles.

Considérons par exemple la version simplifiée de la transformation qui réalise la mise à plat d'une machine à état. Les modèles de sortie ne contiennent pas d'états composites.

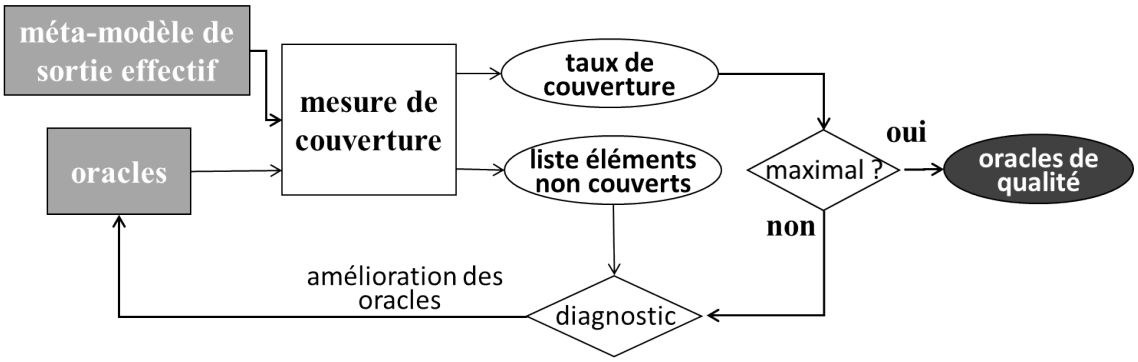


Figure 4.1 – Mesure de la couverture du méta-modèle de sortie effectif par des oracles pour évaluer leur qualité

De ce fait, toute information concernant les états composites est inutile dans le méta-modèle de sortie, nous obtenons ainsi le méta-modèle de sortie effectif de la Figure 4.2.

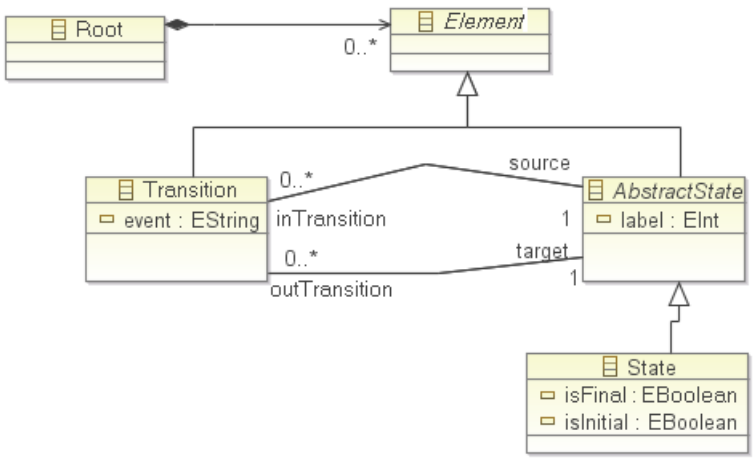


Figure 4.2 – Méta-modèle de sortie effectif pour la version simple de la mise à plat d'une machine à état

Notre proposition ne s'intéresse pas à la transformation sous test. Elle manipule uniquement le méta-modèle de sortie effectif et les oracles de la suite de tests. Notre proposition est donc indépendante du langage de transformation utilisé. Le processus de mesure de la couverture du méta-modèle effectif par les oracles est systématique, et donc automatisable (et automatisé, voir Section 4.2). Notre approche est aussi moins coûteuse que l'analyse de mutation. Le testeur n'a pas à créer ni à exécuter de mutants.

4.1.2 Mesure de la couverture du méta-modèle de sortie

Nous mesurons la couverture du méta-modèle de sortie effectif par un ensemble d'oracles. Dans [Fleurey et al., 2009] Fleurey et al. mesurent la couverture du méta-modèle d'entrée par les modèles de test. Ils ont adapté les critères définis par Andrews et al. [Andrews et al., 2003] pour la couverture de diagrammes de classe UML :

- `Class Coverage` : chaque méta-classe doit être instanciée au moins une fois.
- `Association End Multiplicities` : pour les extrémités de chaque association, chaque multiplicité représentative doit être couverte.
- `Class Attribute` : pour chaque attribut, chaque valeur représentative intéressante pour le testeur doit être couverte.

Nous proposons d'utiliser les mêmes critères, mais nous les adaptons à la couverture du méta-modèle de sortie. Fleurey et al. combinent ces critères pour construire des modèles représentatifs, nous ne le faisons pas. Les modèles d'entrée sont transformés par la transformation sous test. Pour des modèles de test, il est important de combiner les valeurs de leurs éléments. Plusieurs combinaisons différentes en entrée peuvent produire des résultats différents. Ici, nous qualifions des oracles. Les oracles contrôlent les modèles obtenus. Un modèle obtenu n'est pas transformé, c'est un résultat. Dans ce cas, combiner les valeurs n'est pas aussi important. Nous gérons les méta-classes abstraites et les liens d'héritage de la même manière que Fleurey et al. Nous ne traitons pas les méta-classes abstraites. Pour chaque méta-classe concrète, nous vérifions que toutes ses propriétés (attributs et références), propres et héritées sont couvertes. Nous mesurons la couverture d'un méta-modèle par un oracle conformément à la définition suivante :

- un attribut d'une méta-classe est dit couvert s'il est instancié dans l'oracle ;
- une référence est dite couverte si elle est instanciée dans l'oracle ;
- une méta-classe est dite couverte si toutes ses propriétés (attributs et références) sont couvertes.

Dans l'exemple de la mise à plat d'une machine à états hiérarchique, le modèle de couverture est créé à partir du méta-modèle effectif de la Figure 4.2. Un ensemble d'oracles qui couvre la méta-classe `State` contiendra une instance de cette méta-classe. Il contiendra également une instance de chacune des propriétés `label`, `isFinal`, `isInitial`, `inTransition`, et `outTransition`. La méta-classe `AbstractState` est abstraite, nous ne pouvons et ne cherchons pas à la couvrir, nous contrôlons la couverture de ses propriétés dans les instances de sa méta-classe fille. La Figure 4.3 présente un exemple de modèle de machine à états instance du méta-modèle de sortie effectif de la Figure 4.2. Ce modèle contient au moins une instance de chaque méta-classe concrète du méta-modèle effectif, et chacune des propriétés de ces méta-classes est instanciée au moins une fois. Ainsi un oracle utilisant ce modèle comme modèle attendu à comparer au modèle obtenu, couvrirait entièrement le méta-modèle de sortie effectif.

La mesure de la couverture d'un méta-modèle effectif de sortie par un ensemble d'oracles se déroule en deux étapes, comme le montre la Figure 4.4 :

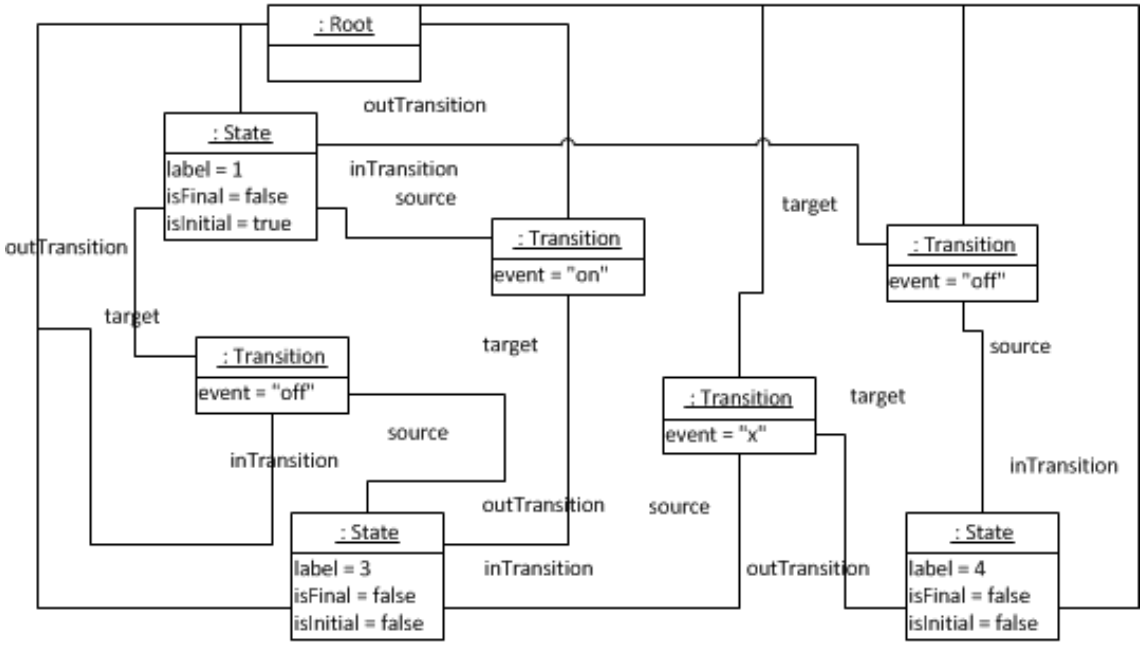


Figure 4.3 – Exemple de modèle de sortie instance du méta-modèle simplifié de machine à état

1. Dans la première étape, nous transformons le méta-modèle de sortie effectif MM_{out} (Figure 4.2, page 4.2) pour ajouter les informations de couverture. Chaque élément à couvrir est annoté. Ce méta-modèle effectif annoté est le modèle de couverture.
2. Dans la seconde étape, le *Coverage Checker* analyse le modèle de couverture et les oracles pour produire deux résultats. Ces résultats sont le taux de couverture calculé et ainsi qu'une liste des éléments non couverts.

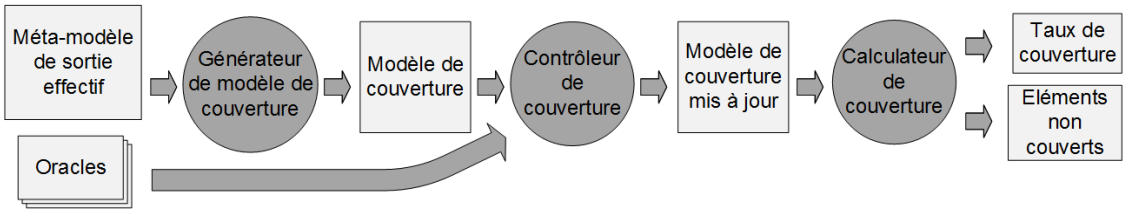


Figure 4.4 – Mesure de la couverture du méta-modèle de sortie par un oracle

4.2 Mise en œuvre de notre proposition

L'approche présentée dans la Section 4.1.1 a été mise en œuvre dans un outil que nous avons développé. Cet outil permet de mesurer la couverture d'un méta-modèle par un ou plusieurs modèles qui lui sont conformes. Dans cette section, nous détaillons cette

mise en œuvre. Après une présentation de l'environnement technique dans lequel nous travaillons, nous détaillons le fonctionnement de notre outil.

4.2.1 Environnement technique

Notre outil permet de mesurer la couverture de méta-modèle au format Ecore par des modèles qui leur sont conformes. La mesure de la couverture d'un méta-modèle se fait à l'aide de plusieurs transformations de modèles successives. Les différentes transformations de cette chaîne ont été la mise en œuvre en Java/EMF.

Dans la seconde étape de notre approche, une transformation mesure la couverture du méta-modèle de sortie effectif par un modèle utilisé comme donnée d'oracle. Pour écrire cette transformation en ATL ou Kermeta il faut connaître par avance les méta-modèles auxquels les modèles manipulés sont conformes. Le méta-modèle effectif est conforme au méta-modèle Ecore, et la donnée d'oracle est conforme au méta-modèle effectif. Nous devons donc connaître le méta-modèle effectif au moment d'écrire la transformation. De plus cette transformation sera spécifique au méta-modèle effectif. Avec Java/EMF, les éléments d'un modèle peuvent être manipulés comme des objets instances de *EObject*, sans tenir compte du méta-modèle. Nos transformations peuvent être utilisées avec n'importe quel modèle, sans être dépendante du méta-modèle auquel il correspond.

Nous proposons de mesurer la couverture du méta-modèle de sortie effectif de la transformation sous test par les oracles de la suite de tests. Ce méta-modèle effectif peut être obtenu par analyse statique de la mise en œuvre de la transformation sous test. Le Footprint proposé par Jeanneret et al. [Jeanneret et al., 2011] analyse une transformation mise en œuvre en Kermeta. Il retourne le méta-modèle d'entrée ou de sortie dans lequel les éléments non utilisés par la transformation sont supprimés. L'utilisation de cette technique pour obtenir un méta-modèle effectif a un inconvénient majeur. Elle est dépendante de la mise en œuvre de la transformation sous test. Toute erreur dans la mise en œuvre sous test peut se répercuter et produire un méta-modèle effectif erroné. Considérons que la transformation du premier cas d'étude ne modifie pas directement le modèle d'entrée, mais crée le modèle de sortie. Si cette transformation ne donne aucune valeur à la propriété `event` lorsqu'une transition est créée, alors un méta-modèle effectif obtenu à partir de cette transformation ne contiendrait pas la propriété `event` dans la méta-classe `Transition`. Ce méta-modèle de sortie effectif serait donc faux. Il est plus sûr de se baser sur la spécification de la transformation pour en obtenir un méta-modèle effectif. Comme pour l'oracle, le testeur n'est pas influencé par les éventuelles erreurs commises par le développeur.

4.2.2 Couverture d'un méta-modèle par un ensemble de modèles

La mesure de la couverture d'un méta-modèle par un ensemble de modèles se fait en plusieurs étapes. Dans la première étape, une première transformation génère un modèle de couverture, à partir du méta-modèle à couvrir. Dans la deuxième étape, une autre transformation met à jour le modèle de couverture avec les éléments instanciés dans chacun des modèles. Dans la troisième étape, une dernière transformation extrait les informations du modèle de couverture mis à jour et retourne les résultats de la mesure.

Modèle de couverture

La première transformation de la chaîne prend en entrée un méta-modèle au format Ecore et produit un modèle de couverture. Ce modèle de couverture est le méta-modèle fourni en entrée dans lequel des annotations sont ajoutées sur les éléments à couvrir par nos oracles :

- les méta-classes non abstraites ;
- les attributs contenus dans ces méta-classes ;
- les références contenues dans ces méta-classes.

Les éventuels liens d'héritage présents dans le méta-modèle sont gérés de manière similaire à ce que proposent Fleurey et al. [Fleurey et al., 2009]. Pour chaque méta-classe concrète, nous contrôlons la couverture de tous ses attributs et références propres et hérités. Un élément est considéré comme couvert s'il est instancié dans au moins un oracle. Par défaut aucun élément n'est couvert.

Nous annotons les éléments du méta-modèle pour indiquer s'ils sont couverts ou non. Chaque annotation possède un attribut source de type chaîne de caractères qui fait office d'identifiant. Le contenu de l'annotation (ou détail) est stocké sous forme de couples clé $\mapsto$ valeur.

Génération du modèle de couverture

La première transformation est en charge de la génération du modèle de couverture (cf Figure 4.5). Elle se déroule comme suit.

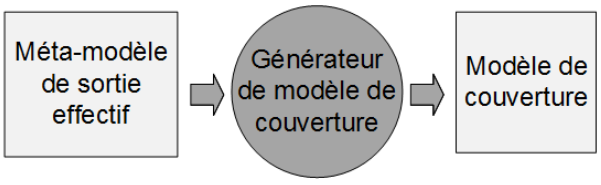


Figure 4.5 – Génération du modèle de couverture

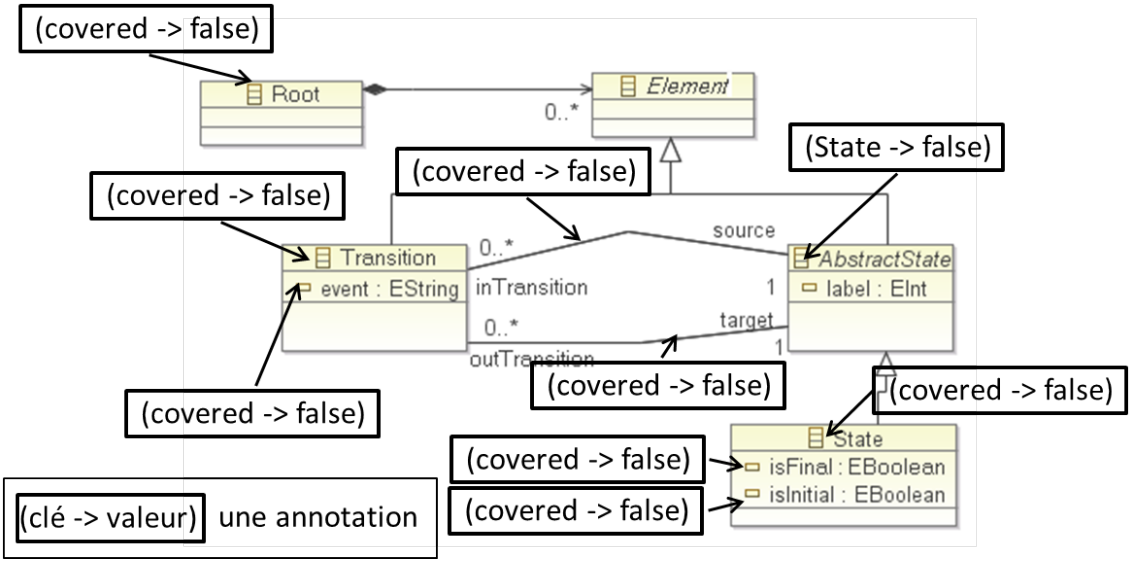


Figure 4.6 – Exemple de modèle de couverture

Pour chaque méta-classe non abstraite contenues dans le *package* du méta-modèle : Faire

1. ajouter une annotation *"coverage"*, qui contient le couple *"covered"* $\mapsto$ *"false"* ;
2. pour chaque propriété (propre et héritée) de cette méta-classe :
   (a) ajouter une annotation *"coverage"* si elle n'est pas déjà présente ;
   (b) ajouter à l'annotation *"coverage"*, le couple *nom de la méta-classe* $\mapsto$ *"false"*.

Considérons le modèle de couverture généré à partir du méta-modèle effectif de la Figure 4.2. Dans le modèle de couverture généré, présenté Figure 4.6, la méta-classe `State` portera une annotation "coverage" avec le couple "covered" $\mapsto$ "false" , et les attributs qu'elle contient (`isInitial` et `isFinal`) porteront une annotation similaire. En revanche sa méta-classe mère, `AbstractState`, ne portera aucune annotation. Ses propriétés, elles, porteront chacune une annotation "coverage" contenant le couple "State" $\mapsto$ "false" correspondant à l'unique méta-classe fille concrète.

Évaluation de la couverture d'un méta-modèle par un modèle

La deuxième transformation permet d'évaluer la couverture du méta-modèle par un oracle (cf Figure 4.7). Elle prend en entrée un modèle de couverture et un modèle (la donnée d'oracle). Elle met à jour le modèle de couverture généré par la transformation précédente en fonction de l'oracle. Elle indique quels éléments du méta-modèle sont instanciés dans l'oracle. Elle est exécutée pour chaque oracle de la suite de tests. Le modèle de couverture mis à jour par une exécution de cette transformation est utilisé pour la suivante.

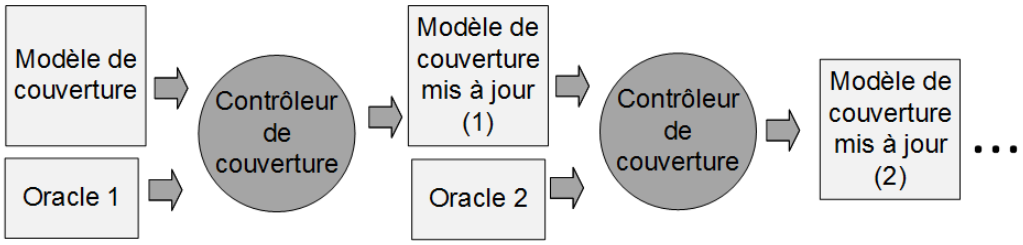


Figure 4.7 – Mise à jour du modèle de couverture

La transformation parcourt les éléments du modèle à évaluer.

Pour chaque objet : Faire

1. rechercher dans le modèle de couverture la méta-classe dont cet objet est instance ;
2. si cette méta-classe n'est pas encore couverte, alors traiter chaque propriété de l'objet courant : la transformation indique que la référence ou l'attribut qui lui correspond dans le modèle de couverture est couvert ;
3. si toutes les propriétés de la méta-classe sont couvertes, alors la méta-classe est couverte elle aussi.

La sortie de cette transformation est le modèle de couverture mis à jour.

Mesure de la couverture

La dernière transformation, effectue la mesure du taux de couverture du méta-modèle à partir d'un modèle de couverture (cf Figure 4.8).

Ce traitement se fait comme suit :

1. parcourir le modèle de couverture et compter le nombre de méta-classes non abstraites. Deux compteurs sont utilisés l'un compte le nombre total de méta-classes non abstraites et l'autre uniquement celles qui ne sont pas couvertes. Le nombre de propriétés est aussi comptabilisé (encore une fois nombre total et non couvertes) ;
2. afficher le taux de couverture du méta-modèle sur la sortie standard. Le taux de couverture est exprimé en terme de méta-classes et propriétés. Afficher la liste des méta-classes et propriétés non couvertes.

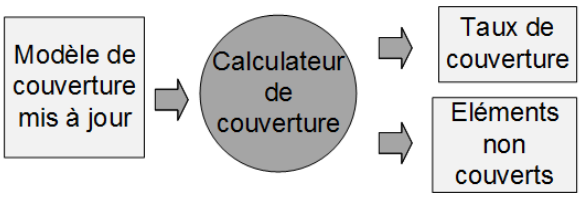


Figure 4.8 – Mesure de la couverture

4.3 Validation expérimentale

Nous avons proposé une approche pour qualifier un oracle pour le test de transformations de modèles, dans cette section nous la validons sur nos deux cas d'étude, présentés dans la Section 1.2. Pour la mise à plat de la machine à états nous utilisons la mise en œuvre basée sur le méta-modèle simplifié ; nous utilisons cette mise en œuvre car ce méta-modèle simplifié est proche d'un méta-modèle effectif. Tout d'abord nous détaillons le protocole d'expérimentation mis en place, puis nous présentons les résultats obtenus et enfin, nous discutons de ces résultats. Le matériel expérimental utilisé est disponible en ligne [Finot et al.,].

4.3.1 Protocole d'expérimentation

Nous avons proposé une nouvelle approche pour qualifier les oracles d'une suite de tests de transformations de modèles. Pour valider cette approche, nous l'appliquons à nos deux cas d'étude et comparons ses résultats avec ceux de l'analyse de mutation.

- nous définissons le méta-modèle de sortie effectif de chaque transformation sous test ;
- nous créons un ensemble de mutants pour les transformations sous test ;
- nous créons un ensemble de modèles ;
- nous créons un ensemble d'oracles correspondant aux modèles de test ;
- nous utilisons notre outil pour mesurer la couverture de ce méta-modèle effectif par les oracles ;
- nous utilisons les oracles pour tuer les mutants et nous calculons le score de mutation pour chaque cas d'étude.

4.3.1.1 Création des modèles de test et des mutants des transformations sous test

Pour la transformation de mise à plat d'une machine à états, nous appliquons à nouveau la stratégie IFClassΣ et obtenons onze fragments de modèles. Nous créons vingt-deux modèles de test à partir de ces fragments, chaque fragment donne deux modèles :

- un modèle avec cinq états, dont au moins un composite et dix transitions ;
- un modèle avec dix états, dont au moins un composite et quinze transitions.

Nous appliquons les opérateurs proposés par Mottu et al. [Mottu et al., 2006a] pour créer les mutants des transformations sous test. Nous créons quatre-vingt-trois mutants pour la première transformation et cent-trente-sept pour la seconde.

Les modèles de test permettent de tuer tous ces mutants, donc toutes les fautes injectées sont mises en évidence. Pour la seconde transformation, nous utilisons les modèles créés lors de nos expériences précédentes (cf Section 3.3.1, page 60).

4.3.1.2 Création des oracles

Pour chaque cas d'étude, nous créons deux ensembles d'oracles (**E1** et **E2**) en utilisant deux des fonctions d'oracles existantes [Mottu, 2008].

Les oracles de l'ensemble **E1** utilisent des modèles attendus qui sont comparés à ceux produits par la transformation sous test. Pour valider notre approche nous avons besoin d'utiliser plusieurs sous-ensembles d'oracles ayant une couverture différente du méta-modèle de sortie effectif. Nous créons donc tout d'abord des oracles partiels selon l'approche proposée dans le chapitre 3, puis nous augmentons la couverture du méta-modèle de sortie effectif pour obtenir une couverture totale. Nos oracles partiels sont créés en ignorant de manière systématique chaque méta-classe, chaque attribut, et chaque attribut du méta-modèle effectif.

De cette manière, nous créons neuf sous-ensembles d'oracles (**P0**, **P1**, **P2**, **P3**, **P4**, **P5**, **P6**, **P7**, **P8**), dont 8 sont partiels, pour la transformation de mise à plat de la machine à états :

- les oracles de l'ensemble **P1** ignorent donc toutes les instances de la méta-classe `State` (voir Figure 4.9, ce sous-ensemble d'oracles ne couvre pas les éléments grisés) ;
- les oracles de l'ensemble **P2** ignorent les instances de la méta-classe `Transition` ;
- les oracles de l'ensemble **P3** ignorent les instances des références `outTransition` et `target` (voir Figure 4.10) ;
- les oracles de l'ensemble **P4** ignorent les instances des références `inTransition` et `source` ;
- les oracles de l'ensemble **P5** ignorent les instances de l'attribut `event` de la méta-classe `Transition` ;
- les oracles de l'ensemble **P6** ignorent les instances de l'attribut `label` de la méta-classe `AbstractState` ;
- les oracles de l'ensemble **P7** ignorent les instances de l'attribut `isFinal` de la méta-classe `State` ;
- les oracles de l'ensemble **P8** ignorent les instances de l'attribut `isInitial` de la méta-classe `State` ;
- les oracles de l'ensemble **P0** n'ignorent aucun élément.

L'ensemble d'oracles **E2** utilise des contrats. Ces contrats expriment des propriétés sur les modèles de sortie, ou alors entre les modèles de test et les modèles de sorties qui leur correspondent, comme ceux utilisés par Cariou et al. [Cariou et al., 2009]. Nous écrivons des contrats individuels. Chacun correspond à une exigence de la spécification que nous souhaitons contrôler. Par exemple, un contrat va vérifier qu'il n'y a aucun état composite dans le résultat de la transformation de mise à plat d'une machine à états. Nous combinons ces contrats individuels pour en construire de nouveaux, de manière incrémentale. Chaque contrat incrémental est la conjonction de plusieurs contrats individuels. Nous commençons par choisir un contrat ayant la couverture du méta-modèle de sortie la

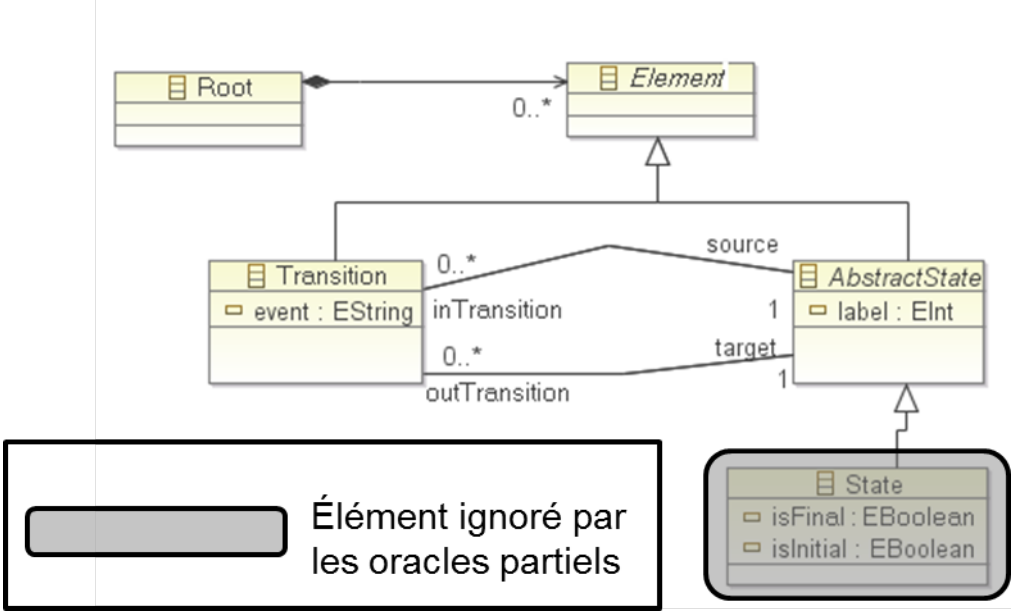


Figure 4.9 – Éléments couverts par l'ensemble d'oracles P1

plus faible. Ensuite, nous ajoutons l'un après l'autre chacun des autres contrats. À chaque étape, nous ajoutons un contrat qui entrainera une augmentation de la couverture la plus petite possible.

Pour la transformation de mise à plat d'une machine à états, six contrats individuels (**C1**, **C2**, **C3**, **C4**, **C5**, **C6**) on été nécessaires :

- le contrat **C1** vérifie qu'il n'y a plus d'états finaux en sortie de la transformation ;
- le contrat **C2** vérifie que les états finaux appartenant à un état composite du modèle d'entrée ne le sont plus en sortie ;
- le contrat **C3** vérifie que les états initiaux appartenant à un état composite du modèle d'entrée ne le sont plus en sortie ;
- le contrat **C4** vérifie dans le listing 4.1 contrôle que les transitions entre états simples sont conservées ;
- le contrat **C5** vérifie que les états simples présents dans le modèle d'entrée le sont toujours dans le modèle de sortie ;
- le contrat **C6** vérifie que les transitions du modèle d'entrée dont une extrémité est un état composite sont correctement transformées.

Comme le montre la Figure 4.12 nous combinons ces contrats individuels afin d'obtenir la meilleure couverture possible du méta-modèle.

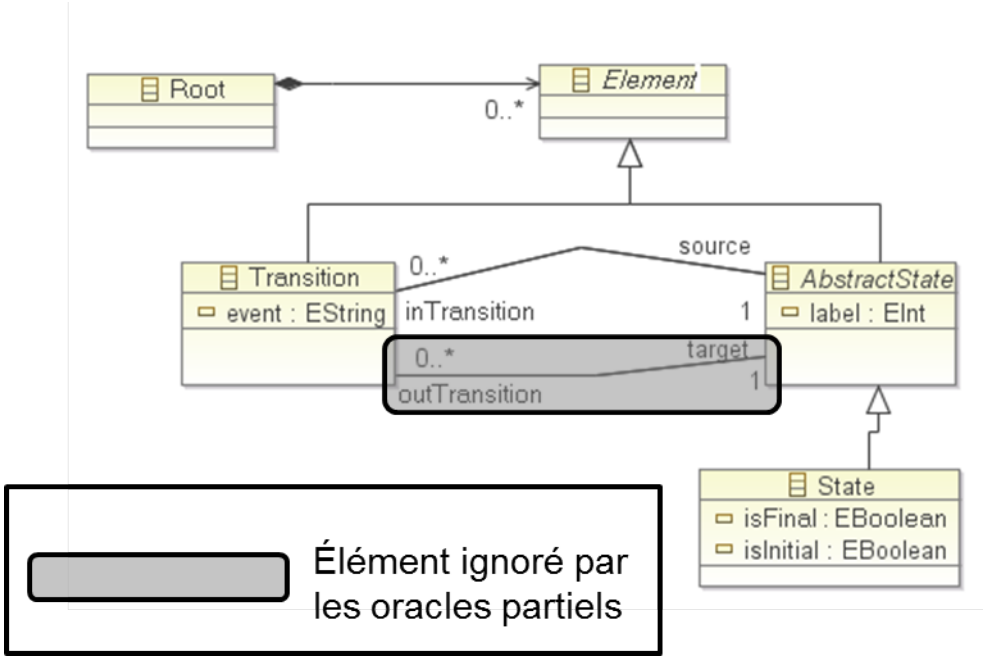


Figure 4.10 – Éléments couverts par l'ensemble d'oracles P3

```
// Simple states are kept
operation keepSimples(input : Root, output : Root) : Boolean is
do
        result := getAllStates(input).collect{s | s.label} == getAllStates(output).
            collect{s | s.label}
end

operation getAllStates(root : Root) : Collection<State> is
do
        result := Set<State>.new

        root.ownedElements
                .select{s | s.isInstanceOf(State)}
                .each{s | do result.add(s.asType(State)) end}
end
```

Listing 4.1 – Un exemple de contrat individuel, le contrat C4

Après cela, nous mesurons la couverture du méta-modèle de sortie par chaque ensemble d'oracles.

4.3.1.3 Analyse de mutation

Nous utilisons la démarche proposée par Mottu et al. [Mottu et al., 2006b] pour qualifier nos oracles à l'aide de l'analyse de mutation ; elle consiste en plusieurs étapes.

Étape 1 : transformation des modèles de test par la transformation sous test ;

Étape 2 : création manuelle des mutants de la transformation sous test en injectant une erreur pour chaque mutant (utilisation des opérateurs proposés par Mottu et al. [Mottu et al., 2006a]) ;
Étape 3 : transformation des modèles de test par les mutants ;
Étape 4 : comparaison des modèles de sortie obtenus avec ceux produits par la transformation sous test ; lorsque les deux modèles de sortie comparés sont différents, le mutant est tué ;
Étape 5 : élimination des mutants équivalents à la transformation sous test ;
Étape 6 : calcul du score de mutation.

Nous obtenons 83 mutants pour la première transformation (`fsm2ffsm`) et 137 pour la seconde (`uml2csp`). Les modèles de test permettent de tuer tous les mutants.

Pour évaluer nos oracles, nous reprenons les modèles de sortie produits par les mutants tués. Les oracles contrôlent ensuite ces mutants (pour chaque cas de test, nous créons plusieurs oracles, cf. Section 4.3.1.2). Un nouveau score de mutation est calculé pour chaque ensemble d'oracles. Si le test échoue, le mutant est tué. L'oracle détecte une erreur dans le modèle de sortie produit par le mutant. Le score de mutation obtenu donne une indication de la qualité de l'ensemble d'oracles.

4.3.2 Résultats et discussion

Notre approche est une alternative à l'analyse de mutation pour évaluer la qualité d'un ensemble d'oracles. Pour la valider nous devons répondre aux questions suivantes :

1. Pouvons-nous mesurer la couverture du méta-modèle de sortie par un oracle ou un ensemble d'oracles ?
2. La couverture du méta-modèle est-elle corrélée au score de mutation ?
3. Un taux de couverture de 100 % indique-t-il que l'ensemble d'oracles est sain pour la transformation sous test ?
4. Est-ce que la couverture du méta-modèle de sortie est un bon indicateur de la qualité d'un ensemble d'oracles ?

Nous détaillons tout d'abord les résultats obtenus. Ensuite à partir de ces résultats nous répondons aux questions posées.

4.3.2.1 Présentation des résultats obtenus

Mise à plat de la machine à états Le graphique de la Figure 4.11 représente les résultats obtenus pour le premier cas d'étude, pour les ensembles d'oracles utilisant des modèles attendus. Chaque point correspond à un ensemble d'oracles. Pour chaque ensemble

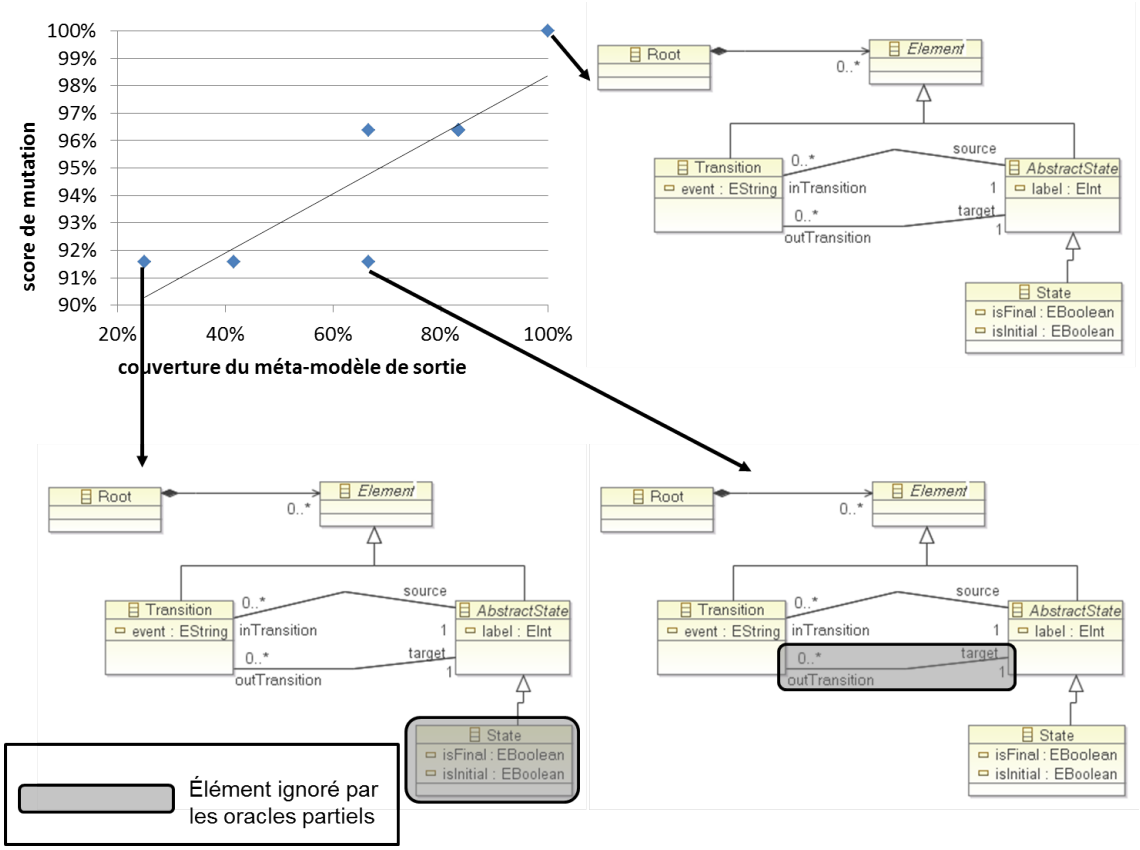


Figure 4.11 – Taux de couverture et score de mutation pour la transformation `fsm2ffsm` (modèles attendus)

d'oracles, son abscisse est son taux de couverture du méta-modèle de sortie effectif, et son ordonnée est son score de mutation. La courbe est la courbe de tendance calculée à partir de ces points.

Cet ensemble contient neuf sous-ensembles d'oracles. Parmi eux, quatre ont un taux de couverture identique et tuent le même nombre de mutants ; ils ne sont représentés que par un seul point. Nous observons sur ce graphique que lorsque le taux de couverture augmente, le score de mutation augmente lui aussi. Cette augmentation est nette sur la courbe de tendance, elle est strictement croissante. Les oracles qui couvrent entièrement le méta-modèle effectif tuent tous les mutants.

Les graphiques des Figures 4.12 et 4.13 représentent les résultats obtenus avec les oracles à base de contrats pour le même cas d'étude. Dans la Figure 4.12 chaque point représente un contrat individuel qui contrôle une exigence de la spécification de la transformation sous test. Dans la Figure 4.13 les points représentent les contrats créés de manière incrémentale. Les deux ensembles de contrats en contiennent chacun huit. Dans les deux cas, deux paires de contrats ont un taux de couverture identiques et tuent le même nombre de mutants. Les courbes sont des courbes de tendance calculées à partir des

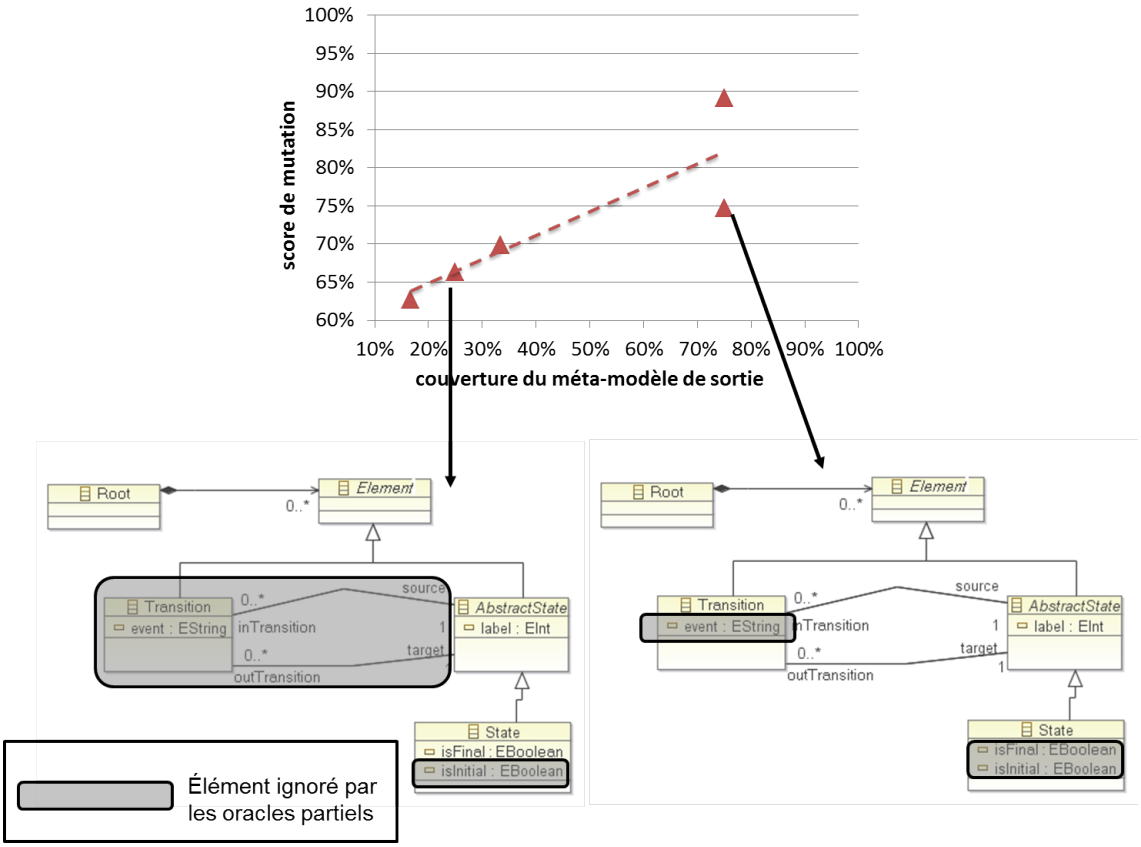


Figure 4.12 – Taux de couverture et score de mutation pour la transformation `fsm2ffsm` (contrats simples)

points. Nous observons sur ces graphiques que les oracles qui ont une meilleure couverture du méta-modèle effectif tuent plus de mutants. La courbe de tendance est strictement croissante. Les contrats incrémentaux parviennent à couvrir entièrement le méta-modèle effectif, mais ne tuent pas tous les mutants.

UML vers CSP Le graphique de la Figure 4.14 représente une partie des résultats obtenus pour la transformation UML vers CSP. Ces résultats concernent les oracles qui utilisent des modèles attendus. Cet ensemble contient seize sous-ensembles d'oracles, parmi eux cinq ont un taux de couverture identique et tuent le même nombre de mutants. Cette fois encore la courbe de tendance est strictement croissante, donc lorsque la couverture du méta-modèle effectif par les oracles augmente, ces oracles tuent plus de mutants. Les oracles qui couvrent entièrement le méta-modèle effectif tuent tous les mutants.

Les graphiques des Figures 4.15(a) et 4.15(b) représentent le reste des résultats obtenus pour le même cas d'étude. La Figure 4.15(a) concerne les contrats individuels. La Figure 4.15(b) concerne les contrats incrémentaux. Pour les contrats individuels, nous observons qu'il n'y a pas une augmentation nette du score de mutation avec l'augmen-

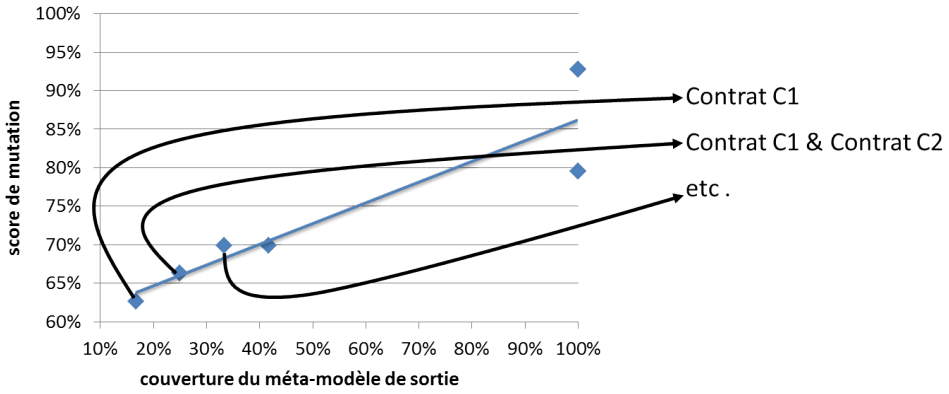


Figure 4.13 – Taux de couverture et score de mutation pour la transformation `fsm2ffsm` (contrats incrémentaux)

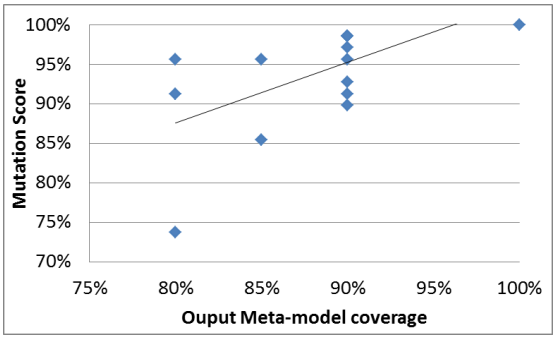
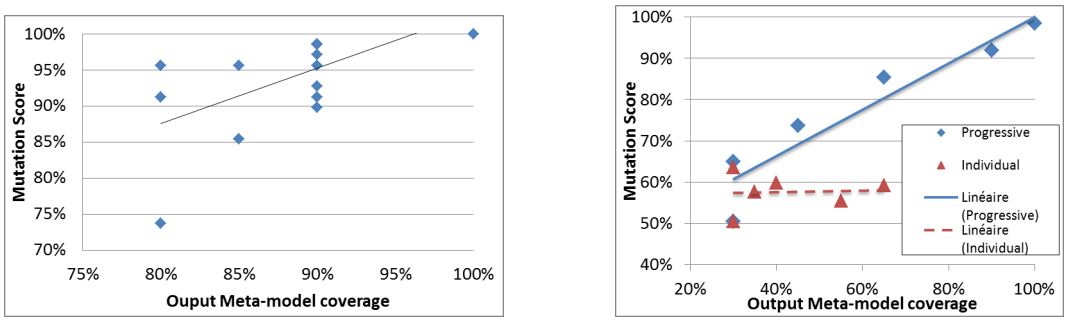


Figure 4.14 – Taux de couverture et score de mutation pour la transformation `uml2csp` (modèles attendus)

tation du taux de couverture du méta-modèle effectif. Cependant, la courbe de tendance est strictement croissante. Pour les contrats incrémentaux, nous observons que la courbe de tendance est strictement croissante. Le contrat incrémental qui couvre entièrement le méta-modèle effectif tue tous les mutants.



(a) contrats individuels (b) contrats incrémentaux

Figure 4.15 – Taux de couverture et score de mutation pour la transformation `uml2csp` (contrats)

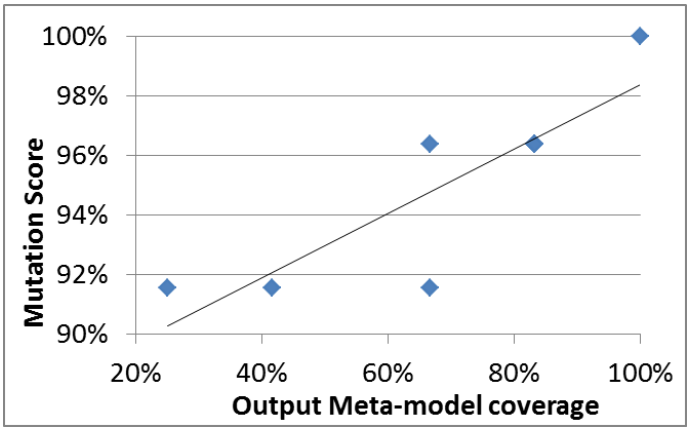
4.3.2.2 Discussion

Nous utilisons à présent les résultats présentés ci-dessus pour répondre aux quatre questions posées page 85.

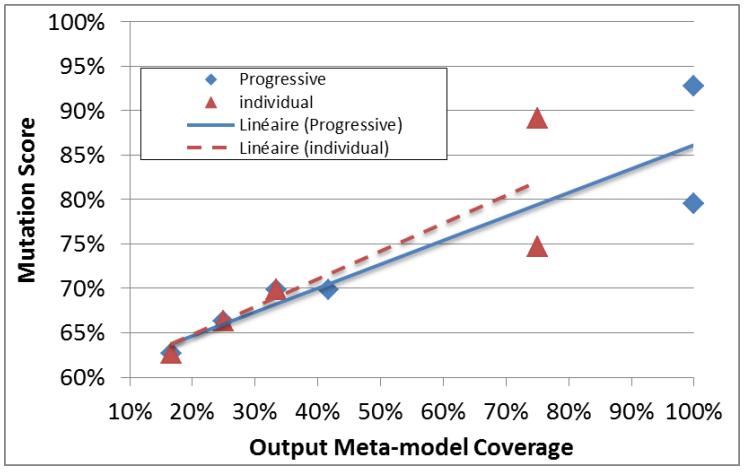
Pour répondre à la première question, nous avons implanté un outil pour mesurer la couverture d'un méta-modèle par un ensemble d'oracles. Nous avons utilisé cet outil sur deux cas d'étude ; Nous avons mesuré la couverture du méta-modèle de sortie effectif par des ensembles d'oracles.

Les Figures 4.16 et 4.17 reprennent les résultats des expériences sur les deux cas d'étude. Les Figures 4.16(a) et 4.17(a) concernent les oracles utilisant une comparaison avec un modèle attendu. Les Figures 4.16(b) et 4.17(b) concernent les contrats. Les points de forme triangulaire et les courbes de tendance pointillées représentent les contrats individuels. Les points de forme carrée et les courbes de tendances pleines représentent les contrats incrémentaux.

Pour répondre à la deuxième question, nous observons que les ensembles d'oracles qui tuent le moins de mutants sont ceux qui ont le taux de couverture le plus faible. À l'inverse, les ensembles d'oracles qui tuent le plus de mutants ont le plus souvent le meilleur taux de couverture. Les contrats individuels de la Figure 4.17(b) sont la seule exception à cette observation. Cependant, nous observons également que toutes les courbes de tendance, sans exception, sont strictement croissantes. Le score de mutation augmente lorsque le taux de couverture du méta-modèle de sortie effectif augmente. Ces expériences montrent donc que ces deux mesures sont corrélées.

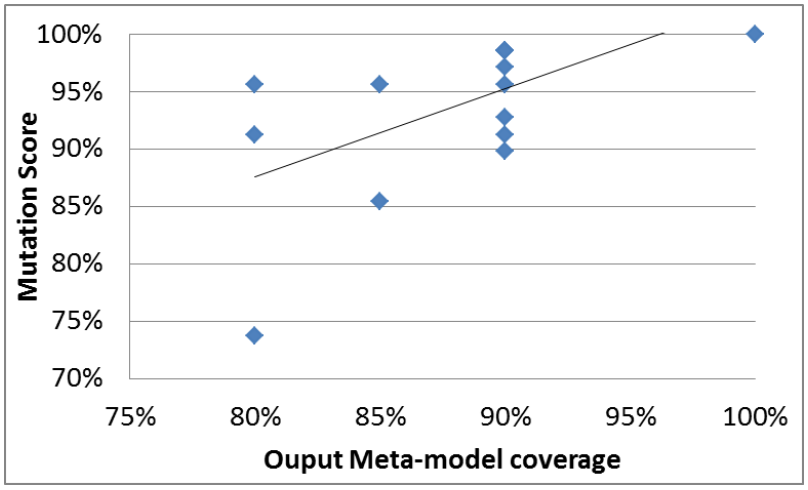


(a) Oracles utilisant des modèles attendus

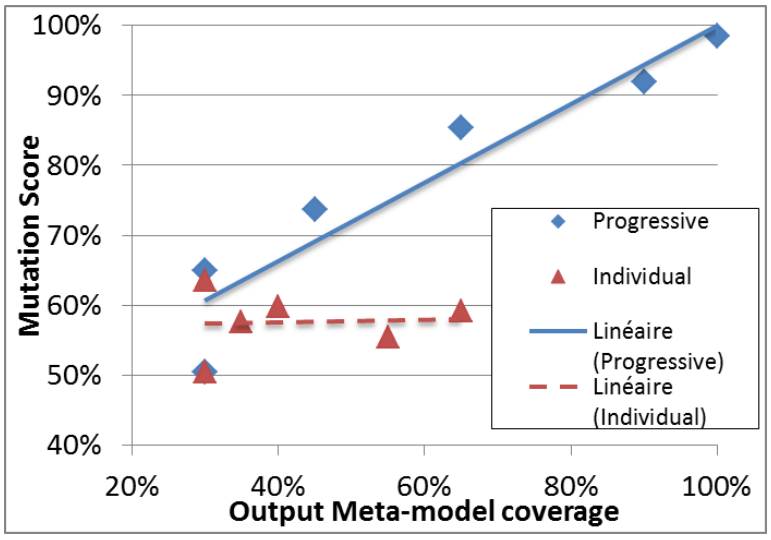


(b) Oracles utilisant des contrats

Figure 4.16 – Lien entre score de mutation et taux de couverture pour la transformation `fsm2ffsm`



(a) Oracles utilisant des modèles attendus



(b) Oracles utilisant des contrats

Figure 4.17 – Lien entre score de mutation et taux de couverture pour la transformation `UML2CSP`

Sur la Figure 4.16(a) nous observons que, pour le premier cas d'étude, l'ensemble d'oracles qui couvre entièrement le méta-modèle de sortie effectif tue tous les mutants. Nous pouvons faire la même observation pour le second cas d'étude. Les oracles qui utilisent des modèles attendus et couvrent entièrement le méta-modèle de sortie effectif, tuent tous les mutants (Figure 4.17(a)). Le dernier contrat incrémental couvre entièrement le méta-modèle de sortie effectif et tue lui aussi tous les mutants (Figure 4.17(b)). Nous observons également que pour le premier cas d'étude, plusieurs contrats incrémentaux couvrent entièrement le méta-modèle de sortie effectif mais ne tuent pas tous les mutants. Les fautes présentes dans les mutants encore en vie ne sont pas détectées par ces contrats. Pour répondre à la troisième question, les expériences montrent qu'un taux de couverture de 100 % n'indique pas que l'ensemble d'oracles est sain. D'après Staats et al. [Staats et al., 2011], s'il y a une faute dans le programme sous test, alors l'oracle la détectera.

Pour ces deux cas d'étude, nous sommes capables de mesurer la couverture du méta-modèle de sortie effectif par un ensemble d'oracles. Ces expériences montrent que le taux de couverture et le score de mutation sont corrélés. Plus le taux de couverture du méta-modèle de sortie effectif est élevé et plus les oracles tuent de mutants, et donc plus ils détectent de fautes. Cependant, nous constatons également que couvrir entièrement le méta-modèle de sortie effectif n'est pas suffisant pour que l'ensemble d'oracles soit sain. Les avantages de notre approche sont qu'elle est moins coûteuse que l'analyse de mutation, n'est pas dépendante de la mise en œuvre de la transformation sous test et est automatisée. Cependant, elle n'est pas aussi précise que l'analyse de mutation. Nous observons dans les résultats obtenus qu'un même taux de couverture peut correspondre à plusieurs scores de mutation.

Pour répondre à la quatrième question, ces expériences nous montrent que le taux de couverture du méta-modèle de sortie effectif par un ensemble d'oracles est un bon indicateur de sa qualité.

4.4 Perspectives : Amélioration de la qualité des oracles

Dans ce chapitre, nous avons proposé et validé une méthode pour évaluer la qualité des oracles d'une suite de tests. Nous mesurons la couverture du méta-modèle de sortie effectif par ces oracles. Pour aller plus loin, nous souhaitons améliorer ces oracles. Dans cette section, nous proposons d'augmenter la couverture du méta-modèle de sortie effectif par les oracles pour les améliorer.

4.4.1 Raisons d'une couverture incomplète

Nous proposons d'aider le testeur à améliorer ses oracles en améliorant leur couverture du méta-modèle de sortie effectif.

Le testeur se trouve donc dans un cas où la couverture du méta-modèle de sortie effectif par les oracles est inférieure à 100 %. Nous identifions deux situations correspondant à ce cas de figure :

1. Les éléments non couverts du méta-modèle de sortie ne sont pas manipulés par la transformation sous test (d'après sa spécification). Il s'agit d'un faux-positif. Les éléments ne devraient pas être considérés pour la mesure de couverture.
2. Les éléments non couverts du méta-modèle sont bien manipulés par la transformation sous test (d'après sa spécification), mais ils ne sont pas contrôlés par les oracles.

La première situation est la raison pour laquelle nous proposons de mesurer la couverture du méta-modèle de sortie effectif par les oracles. Pour la seconde situation, nous distinguons trois cas de figure :

1. les oracles sont insuffisants ;
2. les modèles de tests sont insuffisants ;
3. la transformation sous test est erronée.

A partir de ce constat, nous proposons d'aider le testeur en lui fournissant le moyen d'identifier le cas de figure dans lequel il se trouve.

4.4.2 Rétroaction pour le testeur et aide au diagnostic

Nous avons identifié ci-dessus plusieurs situations où la couverture du méta-modèle de sortie effectif par les oracles de la suite de tests est inférieure à 100 %. Pour que le testeur puisse améliorer la couverture du méta-modèle de sortie effectif, il doit connaître la situation dans laquelle il se trouve. C'est ce que nous proposons de faire dans cette section. Nous utilisons pour cela la liste des éléments du méta-modèle non couverts par les oracles.

Situation 1 : la couverture n'est pas totale car le méta-modèle de sortie utilisé pour la mesure de couverture n'est pas effectif. Pour savoir s'il se trouve dans cette situation, le testeur doit vérifier si les éléments non couverts par ses oracles sont manipulés par la transformation. Pour chaque élément non couvert le testeur vérifie à partir de la spécification si la transformation sous test est sensée le manipuler. Si ce n'est pas le cas, le testeur met à jour le méta-modèle de sortie effectif en retirant cet élément. Après mise à jour il mesure la couverture du nouveau méta-modèle de sortie effectif par ses oracles, en utilisant la méthode décrite dans la section 4.1.2.

Si le problème ne vient pas du méta-modèle effectif, alors le testeur se trouve dans une des deux situations restantes.

Situation 2 : les oracles sont insuffisants. Si le testeur est dans cette situation alors les éléments non couverts par les oracles sont instanciés dans les modèles obtenus. Il peut donc comparer la couverture du méta-modèle de sortie effectif par les modèles obtenus et par les oracles. Si les modèles obtenus couvrent des éléments que les oracles ne couvrent pas, les oracles sont insuffisants. Dans ce cas, le testeur doit compléter les oracles pour qu'ils contrôlent les éléments non couverts.

Situation 3 : la transformation sous test ne produit pas d'instances des éléments non couverts par les oracles. Ce problème peut s'expliquer par des modèles de test insuffisants ou des fautes dans la transformation sous test. Dans cette situation, nous proposons d'utiliser des liens de traçabilité pour identifier les manques dans les modèles de test. Nous proposons de relier les éléments du méta-modèle d'entrée à ceux du méta-modèle de sortie effectif. Avec cette information, le testeur peut identifier les éléments du méta-modèle d'entrée qui sont liés aux éléments non couverts du méta-modèle de sortie effectif. Si ces éléments du méta-modèle d'entrée ne sont pas instanciés dans les modèles de test, alors ces modèles de test sont insuffisants. Pour améliorer ses oracles, le testeur doit créer de nouveaux cas de test. Il ajoute de nouveaux modèles de test qui instancient les éléments identifiés dans le méta-modèle d'entrée. Ensuite, il crée les oracles qui correspondent à ces nouveaux modèles de test. Enfin une fois que les tests ajoutés passent, le testeur évalue à nouveau la qualité des oracles.

Si les éléments identifiés dans le méta-modèle d'entrée sont instanciés dans les modèles de test alors le problème vient de la transformation sous test. Si tous les tests passent, alors les oracles sont eux aussi insuffisants. Ils ne contrôlent que les instances des éléments du méta-modèle de sortie effectif manipulés par la transformation sous test. Cette situation nous semble être la plus rare, la même erreur doit être faite lors de la mise en œuvre de l'oracle. C'est pourquoi il est utile de créer l'oracle en boîte noire indépendamment de la mise en œuvre de la transformation sous test.

Si le méta-modèle de sortie effectif n'est pas totalement couvert par les oracles, alors le testeur se trouve dans au moins une de ces situations. Le principe d'identification et de résolution reste inchangé, chaque étape permet d'identifier et de traiter chaque situation l'une après l'autre.

Nous synthétisons cette démarche d'amélioration des oracles par le schéma présenté Figure 4.18.

4.4.3 Limites de l'aide au diagnostic

Une partie de la solution proposée pour améliorer les oracles utilise des liens de traçabilité. Nous proposons d'utiliser des liens entre les méta-modèles d'entrée et de sortie

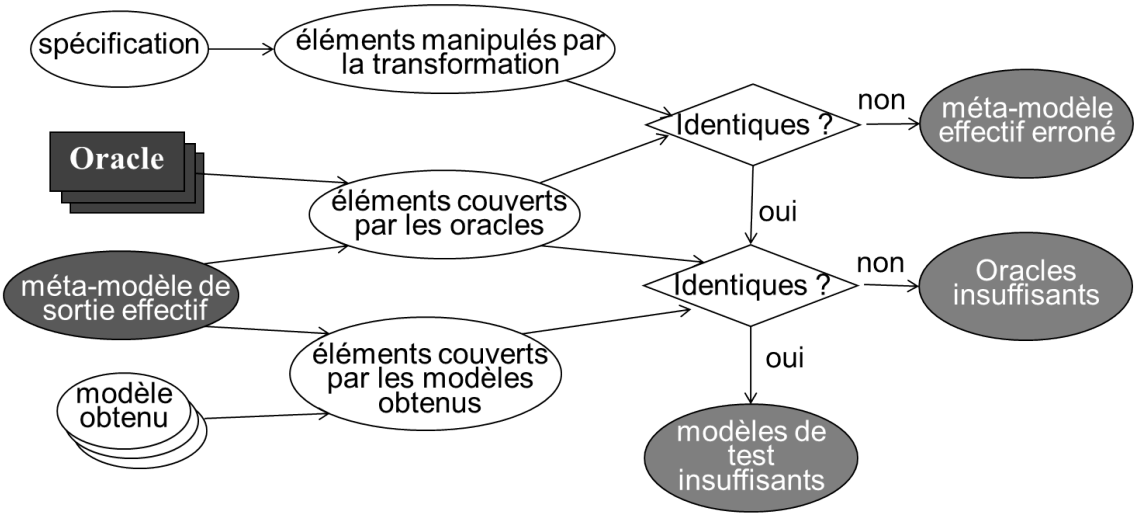


Figure 4.18 – Démarche pour l'amélioration des oracles

d'une transformation pour identifier la source d'une couverture incomplète. Nous considérons ces liens comme des traces statiques.

La principale limite de cette proposition est la définition de ces liens de traçabilité. Tout comme des contrats, ces traces relient l'entrée et la sortie de la transformation sous test. Cependant, elles n'ont pas besoin d'être aussi précises que les contrats. Nous nous intéressons uniquement aux relations entre les éléments et non aux contraintes.

Une première solution pour automatiser cette étape serait de combiner plusieurs traces dynamiques. Dans un premier temps, nous collectons les traces de l'exécution de la transformation sous test sur plusieurs modèles d'entrée (les modèles de test par exemple). Nous combinons ensuite ces traces dynamiques pour définir les liens entre les éléments des méta-modèles. Cette méthode pour obtenir automatiquement ces traces n'est pas très satisfaisante, car nous proposons d'exploiter ces traces statiques pour déterminer si l'ensemble des modèles de test est suffisant. Avec cette méthode, nous obtiendrions des traces qui concernent uniquement les éléments présents dans les modèles d'entrée. S'il manque des éléments dans les modèles d'entrée, ces éléments n'apparaitront pas dans les traces.

Une autre possibilité pour définir automatiquement ces traces statiques, serait d'utiliser l'outil de *footprint* proposé par Jeanneret et al. [Jeanneret et al., 2011]. Cet outil réalise une analyse statique de la transformation sous test. Il retourne une approximation d'un méta-modèle effectif. Les éléments de cette approximation du méta-modèle effectif sont annotés. Ces annotations contiennent le nom des règles de transformations qui manipulent ces éléments. En combinant deux méta-modèles effectifs (entrée et sortie) obtenus avec cet outil nous pourrions obtenir ces traces.

Si nous pouvons utiliser les traces obtenues à l'aide d'un outil tel que le *footprint* pour identifier des éléments du méta-modèle d'entrée qui ne sont pas instanciés dans les

modèles de test, cette méthode n'est pas très satisfaisante. En effet les traces obtenues seraient influencées par les fautes éventuellement présentes dans la transformation sous test. Si un élément du méta-modèle d'entrée n'est, par erreur, pas manipulé par la transformation sous test, alors il n'apparaîtra pas dans les traces. De la même manière que la création d'un oracle se fait en boîte noire, nous pensons que la définition de ces traces statiques devrait se faire à partir de la spécification.

4.5 Conclusion

Dans ce chapitre, nous avons présenté une seconde contribution pour assister le testeur dans le processus de construction d'oracles pour le test de transformations de modèles. Cette approche, mesure la couverture du méta-modèle de sortie effectif par l'ensemble des oracles de la suite de tests, pour les qualifier. Nous évaluons la capacité de ces oracles à détecter des fautes dans la transformation sous test. Plus le taux de couverture est important et meilleurs sont les oracles. Nous avons validé notre approche par des expériences sur deux cas d'étude. Nous comparons les résultats fournis par notre approche à d'autres, obtenus par analyse de mutation. Ces expériences ont montré que les oracles ayant une meilleure couverture du méta-modèle ont tué plus de mutants. Notre approche est automatisée à travers la définition d'un outil, indépendante du langage de transformation utilisé et moins coûteuse que l'analyse de mutation. Cette contribution a fait l'objet d'une publication [Finot et al., 2013b].

En perspective de ce travail, nous proposons d'utiliser la liste des éléments non couverts du méta-modèle de sortie effectif pour améliorer les oracles. Nous proposons d'améliorer la qualité des oracles en améliorant la couverture du méta-modèle de sortie effectif.

CHAPITRE 5

Conclusion et perspectives

Dans cette thèse, nous avons travaillé dans le domaine de l'Ingénierie Dirigée par les Modèles. Cette approche dont l'utilisation est grandissante vise à placer les modèles au cœur du processus de développement logiciel. L'utilisation de modèles permet entre autres de s'abstraire des spécificités d'une plateforme d'exécution pour se concentrer sur le système en lui-même. Les modèles manipulés ne sont pas statiques, ils évoluent au fil de leurs utilisations. Cette évolution se fait par transformation. Les transformations de modèles sont automatisées pour être réutilisées un grand nombre de fois. Avant d'utiliser et de réutiliser une transformation de modèles, l'utilisateur doit lui faire confiance. Il doit être certain que cette transformation est correcte vis-à-vis de sa spécification. Des approches ont été proposées pour la vérification et la validation de transformations de modèles. Parmi ces approches, certaines utilisent des techniques de vérification formelle et d'autres proposent de tester des transformations de modèles.

Au cours de cette thèse, nous avons considéré le test de transformations de modèles. Plus précisément, nous nous sommes intéressés aux problématiques liées à l'oracle du test qui est chargé de produire un verdict. Nous avons proposé et validé plusieurs approches pour assister le testeur dans la création d'un oracle. Nous détaillons ces contributions dans la Section 5.1 Les travaux entrepris dans cette thèse ouvrent plusieurs perspectives que nous présentons dans la Section 5.2.

5.1 Contributions

Concernant le test de transformations de modèles, les travaux de recherche s'articulent autour de deux thèmes principaux : la sélection de modèles de test et la création d'oracles. La génération et la sélection de modèles de test ont été le sujet de beaucoup d'études. La définition d'un oracle pour contrôler les modèles obtenus reste un problème peu exploré. Générer automatiquement un oracle n'est pas possible sans spécification formelle ; cela reste une tâche essentiellement manuelle. L'objectif de cette thèse et d'assister le testeur dans cette tâche. Nous avons répondu aux questions suivantes :

A : Les fonctions d'oracles existantes sont-elles adaptées à tous les besoins du testeur ?
B : Comment évaluer la qualité d'un oracle pour le test de transformations de modèles ?

Nous avons proposé de fournir au testeur une assistance à la fois passive et active. Nous avons fourni une assistance passive pour répondre à la question **A**. Nous avons complété l'ensemble d'outils à la disposition du testeur pour construire des oracles. Nous avons proposé une nouvelle fonction d'oracle qui vient compléter celles qui existent déjà. Avec cette fonction d'oracle, le testeur peut ne contrôler qu'une partie d'un modèle obtenu. Nous avons fourni une assistance active pour répondre à la question **B**. Nous avons proposé une nouvelle approche pour évaluer la qualité des oracles d'une suite de tests. Nous proposons également d'exploiter le résultat de l'évaluation de la qualité des oracles pour les améliorer.

5.1.1 Fonction d'oracle partielle pour le test de transformations de modèles

Dans le chapitre 3, nous avons proposé une nouvelle fonction d'oracle dédiée au test de transformations de modèles. Dans certains cas, le testeur souhaite ne contrôler qu'une partie d'un modèle obtenu. Les fonctions d'oracles existantes ne le permettent pas, pour un coût raisonnable.

Nous avons proposé une nouvelle fonction d'oracle partielle permettant de ne contrôler qu'une partie d'un modèle obtenu. Cette fonction d'oracle compare le modèle obtenu avec un modèle attendu. Elle filtre ensuite le résultat de cette comparaison pour retirer les différences observées à propos d'éléments de la partie qui n'intéresse pas le testeur. Après ce filtrage, si le résultat de cette comparaison de modèles filtrée est vide alors le test passe, sinon il échoue. Le modèle attendu peut être partiel, il s'agit uniquement de la valeur attendue de la partie qui intéresse le testeur. Il peut aussi être entier si le testeur peut le fournir.

Cette nouvelle fonction d'oracle ne remplace pas les fonctions déjà existantes, elle les complète. Cette fonction est utile dans le cas où le testeur souhaite contrôler uniquement une partie d'un modèle obtenu. De plus, nous proposons d'utiliser des contrats en complément de cette fonction d'oracle. Nous utilisons des contrats simples pour compléter le verdict partiel obtenu avec notre fonction d'oracle.

Nous avons validé cette approche en menant des expériences sur deux cas d'études. Ces expériences ont montré qu'un testeur pouvait créer des modèles attendus partiels. Ces modèles attendus partiels peuvent être utilisés par notre fonction d'oracle pour produire un verdict partiel. Avec notre fonction d'oracle partielle, nous avons identifié des erreurs dans la partie contrôlée des modèles obtenus de deux cas d'étude. D'autres fonctions

d'oracle permettent également de détecter ces erreurs dans les modèles de sortie obtenus, mais pour un coût plus élevé.

Cette contribution a fait l'objet de publications à CIEL 2012 [Finot et al., 2012a] au PhD Workshop de ICTSS 2012 [Finot et al., 2012c], et ICMT 2013 [Finot et al., 2013a] ainsi qu'un poster aux journées du GDR GPL 2012 [Finot et al., 2012b].

5.1.2 Qualification d'oracles pour le test de transformations de modèles

Dans le chapitre 3 nous avons proposé une nouvelle fonction d'oracle, complémentaire à celles déjà existantes. Après cela, dans le chapitre 4 nous nous sommes intéressés à l'évaluation de la qualité des oracles d'une suite de tests, et leur amélioration.

Nous considérons la qualité d'un ensemble d'oracles comme leur capacité à détecter des fautes dans la transformation sous test. L'analyse de mutation est une solution utilisée pour qualifier un ensemble d'oracles. Avec cette approche, le testeur crée des mutants, mises en œuvre erronées de la transformation sous test. Il vérifie que ces oracles détectent les erreurs introduites dans les modèles de sortie produits par les mutants. Cependant l'analyse de mutation a quatre inconvénients principaux : elle est coûteuse (i), sa mise en œuvre est principalement manuelle et dépendante du langage de transformation utilisé (ii), et il est difficile d'exploiter ses résultats pour améliorer les oracles qualifiés (iii).

Nous avons proposé d'utiliser une mesure de couverture pour qualifier un ensemble d'oracles. Nous mesurons la couverture du méta-modèle de sortie effectif de la transformation par cet ensemble d'oracles. Le ratio du nombre d'éléments couverts sur le nombre total d'éléments dans le méta-modèle est un indicateur de la qualité de l'ensemble d'oracles. Nous avons également réfléchi à l'amélioration de ces oracles. Nous avons proposé d'augmenter la couverture du méta-modèle de sortie effectif par les oracles pour augmenter leur qualité.

Nous avons validé cette approche par des expériences sur deux cas d'études. Ces expériences ont montré que notre mesure de la couverture du méta-modèle de sortie effectif par un ensemble d'oracles permet d'évaluer la qualité de ces oracles. Notre approche est moins coûteuse à mettre en œuvre et est surtout indépendante du langage de transformation utilisé. De plus, nous pouvons exploiter les résultats de notre mesure pour améliorer la qualité des oracles de la suite de tests.

Cette contribution a fait l'objet d'une publication au workshop AMT 2013 [Finot et al., 2013b].

5.2 Perspectives

Dans cette thèse, notre objectif a été d'assister le testeur dans la création d'oracles pour le test de transformations de modèles. Dans cette section, nous présentons des idées de travaux futurs pour aller plus loin dans cet objectif. D'une part, nous proposons d'approfondir l'évaluation de la qualité des oracles d'une suite de tests. D'autre part, nous proposons de mener une étude sur les différentes fonctions d'oracles existantes.

5.2.1 Approfondir notre évaluation de la qualité

Dans la Section 4 nous avons proposé puis validé une méthode pour évaluer la qualité de l'ensemble des oracles d'une suite de tests. Nous utilisons une mesure de couverture pour évaluer la capacité de ces oracles à détecter des fautes. Pour approfondir cette évaluation, nous proposons dans un premier temps de rechercher d'autres critères de couverture du méta-modèle de sortie effectif. Dans un second temps, nous proposons d'étudier d'autres définitions de la qualité d'un ensemble d'oracles.

5.2.1.1 Autres critères de couverture

Des expériences ont permis de valider notre critère d'évaluation de la qualité des oracles (Section 4.3.2, page 85). Nous avons montré qu'il était possible de substituer la mesure de la couverture à la mesure du score de mutation pour gagner en efficacité. Néanmoins, la précision de cette mesure étant moindre pour qualifier un oracle, nous prévoyons d'étudier d'autres manières de mesurer la couverture.

Nous essaierons d'autres critères de couverture et comparerons les résultats obtenus avec ceux de notre approche. Nous essaierons par exemple de combiner les éléments du méta-modèle de sortie effectif à la manière de Fleurey et al. [Fleurey et al., 2009], pour créer des fragments de modèles. Ces fragments de modèles devront être présents dans les oracles. Tous les fragments ainsi créés ne correspondront pas nécessairement à des modèles de sortie valides d'après la spécification de la transformation sous test. Nous devrons donc éliminer manuellement les fragments invalides, à partir de la spécification de la transformation sous test. Le taux de couverture sera le rapport du nombre de fragments présents dans les oracles sur le nombre total de fragments.

Un avantage de notre approche est la possibilité d'exploiter le résultat de la mesure de couverture pour améliorer les oracles. Avec ce nouveau critère, l'amélioration sera peut-être plus compliquée. Les oracles devront correspondre à plus de fragments. Déterminer la raison de la non-couverture risque d'être plus difficile pour un fragment de modèle que pour un élément.

Nous envisageons également d'améliorer le critère existant en y ajoutant des contraintes. Nous étudierons l'idée de pondérer certains éléments du méta-modèle de sortie effectif de l'une des manières suivantes ou en les combinant :

- donner plus d'importance aux méta-classes qu'aux propriétés,
- donner plus d'importance aux références qu'aux attributs,
- prendre en compte le nombre d'instances de chaque élément dans les oracles,
- prendre en compte le nombre d'utilisations de chaque élément par la transformation sous test.

Avec cette solution, la démarche que nous avons proposée pour améliorer la qualité des oracles sera toujours valide (Section 4.4, page 92).

Si l'une de ces solutions donne de meilleurs résultats, il faudra tout de même prendre en compte sa complexité et le temps d'exécution nécessaire pour sa mise en œuvre. Notre but sera d'améliorer la précision en conservant un coût raisonnable par rapport à l'analyse de mutation.

5.2.1.2 Autres définitions de la qualité d'un ensemble d'oracles

Nous avons défini la qualité d'un ensemble d'oracles comme leur capacité à détecter des fautes dans la transformation sous test. Pour approfondir notre évaluation de la qualité des oracles, nous pourrons considérer d'autres caractéristiques des oracles. Nous chercherons ensuite des critères pour évaluer ces caractéristiques.

Staats et al. [Staats et al., 2011] ont défini plusieurs caractéristiques pour un oracle, il peut, par exemple être complet ou sain. Ces caractéristiques sont liées à la capacité à détecter des fautes dans la transformation sous test ; un oracle est complet s'il n'y a pas de faux positifs. Si un test échoue, alors il y a une faute dans le système sous test. Un oracle est sain s'il n'y a pas de faux négatif. S'il y a une erreur, le test échoue.

Nous étudierons ces propriétés à l'échelle de l'ensemble des oracles de la suite de test. Nos expériences ont démontré que des oracles couvrant la totalité du méta-modèle de sortie effectif ne sont pas forcément sains. Nous rechercherons une métrique qui permettra de déterminer que les oracles sont complets et/ou sains (selon les définitions de Staats et al.). Si avec nos mesures nous parvenons à déterminer qu'un ensemble d'oracles est parfait, nous saurons que ce n'est plus la peine d'essayer d'améliorer ces oracles.

5.2.2 Comparaison des fonctions d'oracles

Nous avons à notre disposition plusieurs fonctions d'oracles ainsi qu'un outil permettant de qualifier des oracles. Après avoir amélioré notre méthode d'évaluation de la qualité d'un oracle ou un ensemble d'oracles, nous la mettrons en pratique.

Pour aller encore plus loin dans l'assistance fournie au testeur, nous comparerons les fonctions d'oracles de manière empirique. Nous écrirons des suites de tests pour plusieurs cas d'étude. Nous créerons plusieurs ensembles d'oracles ; pour chaque cas de test, nous créerons des oracles en utilisant chacune des fonctions d'oracles existantes. Nous évaluerons ensuite leur qualité à l'aide de notre outil, puis nous les améliorerons.

Pour chaque cas d'étude, nous observerons plusieurs informations à propos de ces oracles. Nous considérerons des critères comme le temps nécessaire à leur création ou leur exécution, leur complexité ou encore leur qualité évaluée encore une fois à l'aide de notre outil. Pour chaque cas d'étude, nous dresserons une liste des avantages et inconvénients de chaque fonction d'oracle. De cette manière, selon la transformation sous test, nous pourrons indiquer au testeur la fonction d'oracle la plus adaptée à ses besoins.

5.2.3 Localisation des fautes dans la transformation sous test

Une suite logique à notre travail d'assistance à la création d'oracles sera, à plus long terme, d'aider à localiser les fautes détectées par un oracle de test de transformations de modèles. Détecter la présence de fautes dans la transformation sous test n'est qu'une première étape ; pour corriger ces fautes, le testeur doit les localiser. La localisation des fautes se fera en deux étapes : tout d'abord, nous identifierons les éléments du modèle obtenu concernés par la faute, ensuite nous identifierons la règle de transformation fautive qui produit cet élément.

Si la fonction d'oracle utilisée compare le modèle obtenu à un modèle attendu alors les éléments du modèle obtenu concernés par la faute seront ceux à propos desquels une différence a été observée. Si l'oracle utilise des contrats, alors nous pourrons utiliser notre mesure de couverture. Nous comparerons la couverture du méta-modèle de sortie effectif par le modèle obtenu du cas de test avec la couverture par l'ensemble des modèles de sortie des cas de test pour lesquels le verdict est *passe*. Les éléments concernés par la faute feront partie de ceux couverts uniquement par le modèle obtenu.

Une fois les éléments du modèle de sortie identifiés nous devrons localiser la règle de transformation qui les a produits. Nous pourrons donc utiliser un outil d'analyse statique du code de la transformation sous test, tel que le *footprint* proposé par Jeanneret et al. [Jeanneret et al., 2011]. Cet outil associe chaque élément du méta-modèle aux règles de transformation qui le manipulent. Ainsi, à partir des éléments concernés par la faute, nous pourrons remonter à un ensemble de règles de transformation qui ont produit ces éléments. Le testeur devra alors analyser ces règles pour identifier plus précisément la faute et la corriger.

Annexes

ANNEXE A

Dans cette annexe, nous présentons une partie des données d'oracle utilisée pour tester les transformations présentées dans le chapitre 3.

Sont regroupés :
- Les fragments de méta-modèle que nous avons écrits.
- Les patterns Incquery générés à partir de ces patterns.

A.1 Mise à plat d'une machine à état

Pour la transformation de mise à plat d'une machine à état, la partie non prévisible des modèles de sortie est constituée des états finaux, des transitions ciblant ces états finaux, ainsi que des diverses informations portées par ces transitions.

A.1.1 Fragments de métamodèle

La figure A.1 présente les éléments de la partie non contrôlée des modèles de sortie, à savoir : (a) un état final, (b) une transition ciblant un état final, (c) l'action d'une transition ciblant un état final, (d) une garde portée par une transition ciblant un état final et (e) un trigger porté par une transition ciblant un état final.

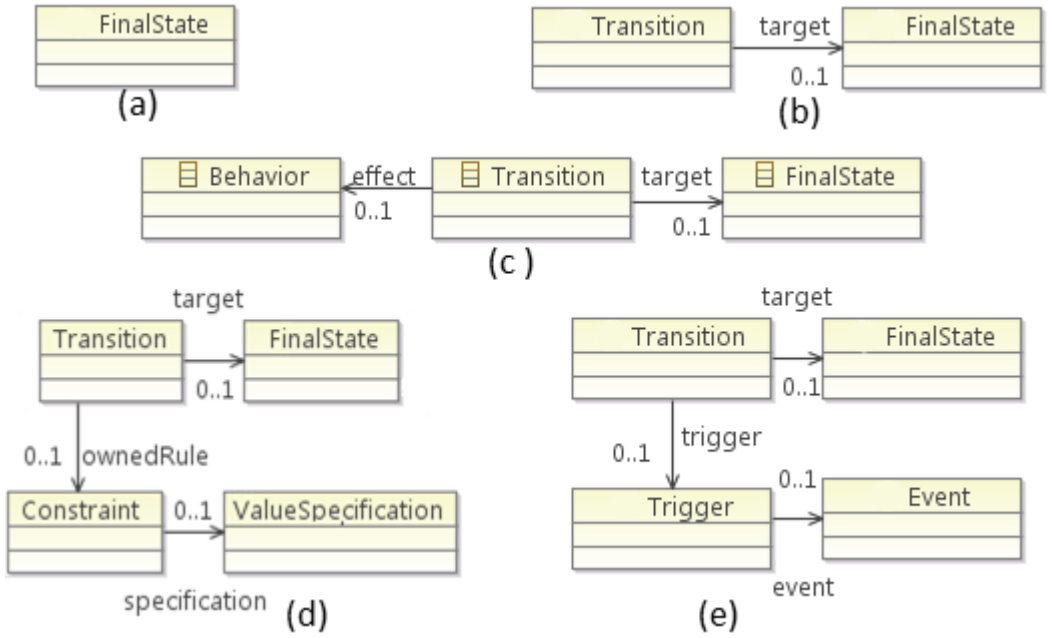


Figure A.1 – Patterns décrivant la partie non prévisible des machines à état mises à plat

A.1.2 Patterns Incquery générés

```
package patterns

import "http://www.eclipse.org/emf/2002/Ecore"
import "http://www.eclipse.org/emf/compare/diff/1.1"
import "http://www.eclipse.org/uml2/3.0.0/UML"

pattern isDifference(E) = {
        ModelElementChangeLeftTarget(ME);
        ModelElementChangeLeftTarget.leftElement(ME, E);
}or{
        ModelElementChangeRightTarget(ME);
        ModelElementChangeRightTarget.rightElement(ME, E);
}or{
        UpdateModelElement(UME);
        UpdateModelElement.leftElement(UME, E);
}or{
        UpdateModelElement(UME);
        UpdateModelElement.rightElement(UME, E);
}or{
        AttributeChange(AC);
        AttributeChange.leftElement(AC, E);
}or{
        AttributeChange(AC);
        AttributeChange.rightElement(AC, E);
}or{
        ReferenceChange(RC);
        ReferenceChange.leftElement(RC, E);
}or{
        ReferenceChange(RC);
        ReferenceChange.rightElement(RC, E);
}
```

```
pattern finalState(C1) = {
        FinalState(C1);
}

pattern finalTransition(C1) = {
        Transition(C1);
        FinalState(C2);
        Transition.target(C1, C2);
}

pattern guard(C4) = {
        Transition(C1);
        FinalState(C2);
        Transition.target(C1, C2);
        Constraint(C3);
        Transition.ownedRule(C1, C3);
        ValueSpecification(C4);
        Constraint.specification(C3, C4);
}

pattern action(C2) = {
        Transition(C1);
        Behavior(C2);
        Transition.effect(C1, C2);
        FinalState(C3);
        Transition.target(C1, C3);
}

pattern trigger(C4) = {
        Transition(C1);
        FinalState(C2);
        Transition.target(C1, C2);
        Trigger(C3);
        Transition.trigger(C1, C3);
        Event(C4);
        Trigger.event(C3, C4);
}

pattern verdictPassIfEmpty(A) = {
        find isDifference(A);
        neg find finalState(A);
        neg find finalTransition(A);
        neg find guard(A);
        neg find action(A);
        neg find trigger(A);
}
```

A.2 UML vers CSP

A.2.1 Fragments de méta-modèle

La partie non contrôlée d'un modèle CSP est décrite par les patterns de la Figure A.2. Elle est composée des opérateurs binaires (a) et de leurs opérandes gauche (b) et droit (c).

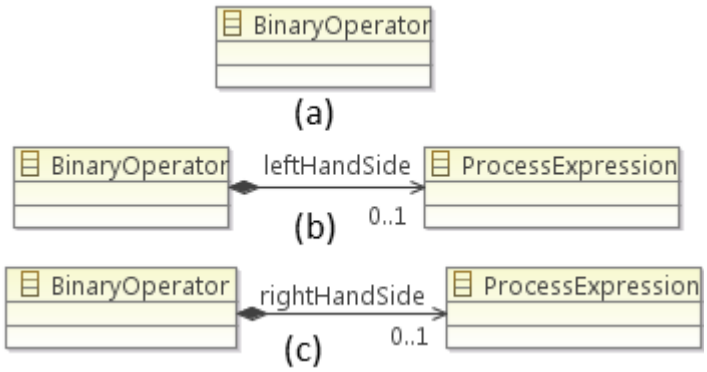


Figure A.2 – Patterns décrivant la partie non contrôlée des modèles CSP

A.2.2 Patterns Incquery générés

```
package patterns

import "http://www.eclipse.org/emf/2002/Ecore"
import "http://www.eclipse.org/emf/compare/diff/1.1"
import "/UML2CSP_old/CSP/CSP.ecore"

pattern isDifference(E) = {
        ModelElementChangeLeftTarget(ME);
        ModelElementChangeLeftTarget.leftElement(ME, E);
}or{
        ModelElementChangeRightTarget(ME);
        ModelElementChangeRightTarget.rightElement(ME, E);
}or{
        UpdateModelElement(UME);
        UpdateModelElement.leftElement(UME, E);
}or{
        UpdateModelElement(UME);
        UpdateModelElement.rightElement(UME, E);
}or{
        AttributeChange(AC);
        AttributeChange.leftElement(AC, E);
}or{
        AttributeChange(AC);
        AttributeChange.rightElement(AC, E);
}or{
        ReferenceChange(RC);
        ReferenceChange.leftElement(RC, E);
}or{
        ReferenceChange(RC);
        ReferenceChange.rightElement(RC, E);
}


pattern leftHandSide(C2) = {
        BinaryOperator(C1);
        ProcessExpression(C2);
        BinaryOperator.leftHandSide(C1, C2);
}

pattern rightHandSide(C2) = {
        BinaryOperator(C1);
        ProcessExpression(C2);
        BinaryOperator.rightHandSide(C1, C2);
```

```
}

pattern binaryOp(C1) = {
        BinaryOperator(C1);
}

pattern verdictPassIfEmpty(A) = {
        find isDifference(A);
        neg find leftHandSide(A);
        neg find rightHandSide(A);
        neg find binaryOp(A);
}
```

ANNEXE B

Dans cette annexe, nous détaillons les différents ensembles d'oracles que nous avons créés pour la validation expérimentale de notre seconde contribution présentée Section 4.3, ainsi que les résultats obtenus.

Pour chaque cas d'étude, nous avons créé plusieurs ensembles d'oracles ; une partie de ces ensembles d'oracles utilisent une comparaison de modèles partielle (cf Chapitre 3), le reste utilise des contrats.

Mise à plat d'une machine à état

Nous créons huit ensembles d'oracles partiels pour la transformation de mise à plat de la machine à états :

1. les oracles de l'ensemble **P1** ignorent toutes les instances de la méta-classe `State`, comme le montre la figure B.1 ;
2. les oracles de l'ensemble **P2** ignorent les instances de la méta-classe `Transition` (figure B.2) ;
3. les oracles de l'ensemble **P3** ignorent les instances des références `outTransition` et `target` (figure B.3) ;
4. les oracles de l'ensemble **P4** ignorent les instances des références `inTransition` et `source` (figure B.4) ;
5. les oracles de l'ensemble **P5** ignorent les instances de l'attribut `event` de la méta-classe `Transition` (figure B.5) ;
6. les oracles de l'ensemble **P6** ignorent les instances de l'attribut `label` de la méta-classe `AbstractState` (figure B.6) ;
7. les oracles de l'ensemble **P7** ignorent les instances de l'attribut `isFinal` de la méta-classe `State` (figure B.7) ;
8. les oracles de l'ensemble **P8** ignorent les instances de l'attribut `isInitial` de la méta-classe `State` (figure B.8) ;
9. enfin les oracles de l'ensemble **P0** n'ignorent aucun élément.

Nous créons également six contrats individuels pour cette transformation :

1. le contrat **C1** exprime qu'il n'y a plus d'état final en sortie de la transformation ;
2. le contrat **C2** (figure B.10) exprime que les états finaux appartenant à un état composite du modèle d'entrée ne le sont plus en sortie ;
3. le contrat **C3** exprime que les états initiaux appartenant à un état composite du modèle d'entrée ne le sont plus en sortie ;
4. le contrat **C4** (figure B.11) exprime que les transitions entre états simples sont conservées ;
5. le contrat **C5** exprime que les états simples présents dans le modèle d'entrée le sont toujours dans le modèle de sortie ;
6. le contrat **C6** exprime que les transitions du modèle d'entrée dont une extrémité est un état composite sont correctement transformées.

Comme le montre la figure B.12 nous combinons ces contrats individuels afin d'obtenir la meilleure couverture possible du méta-modèle.

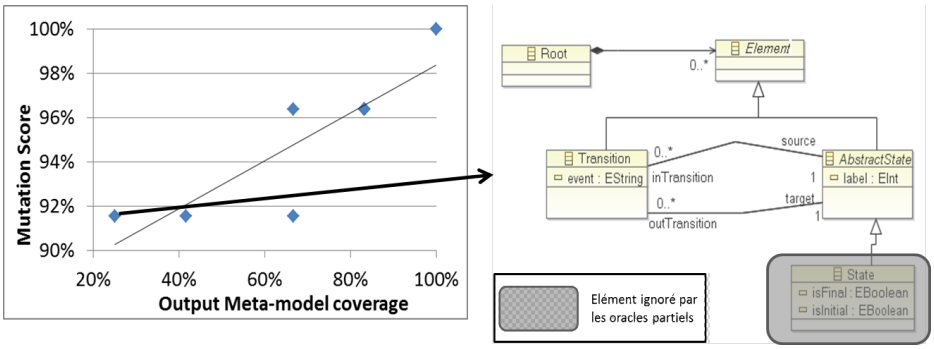


Figure B.1 – Lien entre score de mutation et taux de couverture pour la transformation fsm2ffsm (ensemble P1)

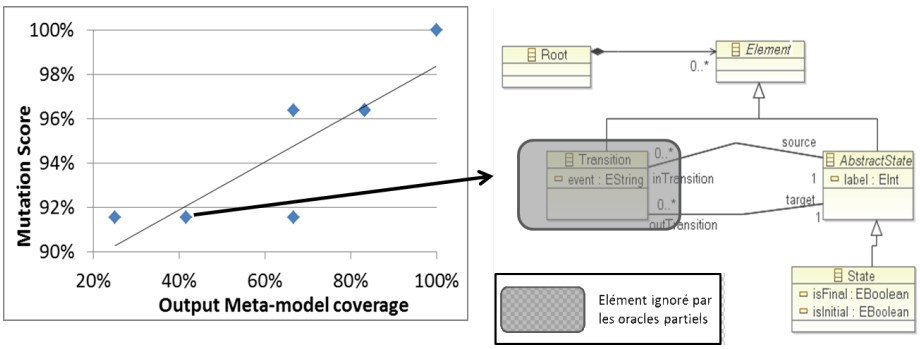


Figure B.2 – Lien entre score de mutation et taux de couverture pour la transformation fsm2ffsm (ensemble P2)

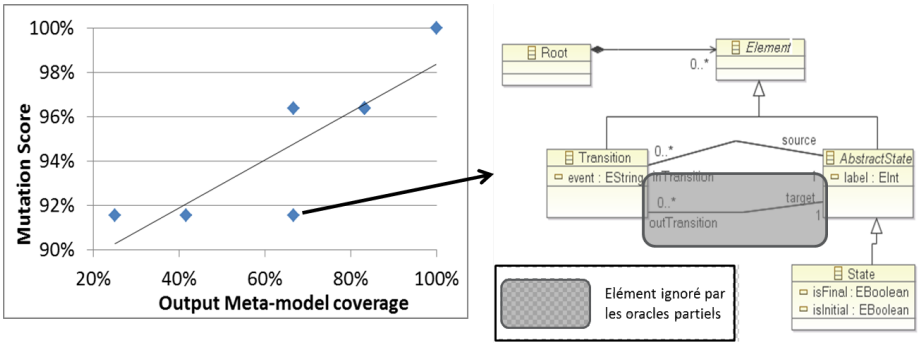


Figure B.3 – Lien entre score de mutation et taux de couverture pour la transformation fsm2ffsm (ensemble P3)

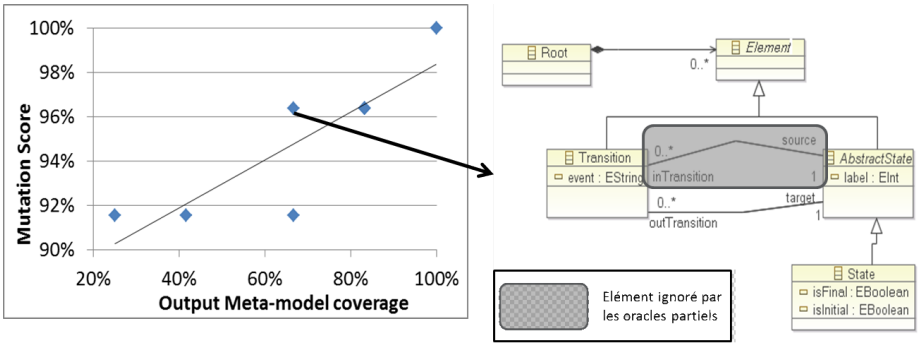


Figure B.4 – Lien entre score de mutation et taux de couverture pour la transformation fsm2ffsm (ensemble P4)

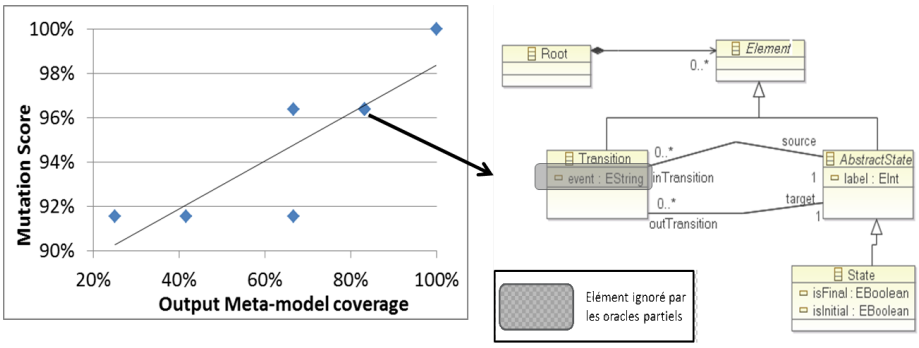


Figure B.5 – Lien entre score de mutation et taux de couverture pour la transformation fsm2ffsm (ensemble P5)

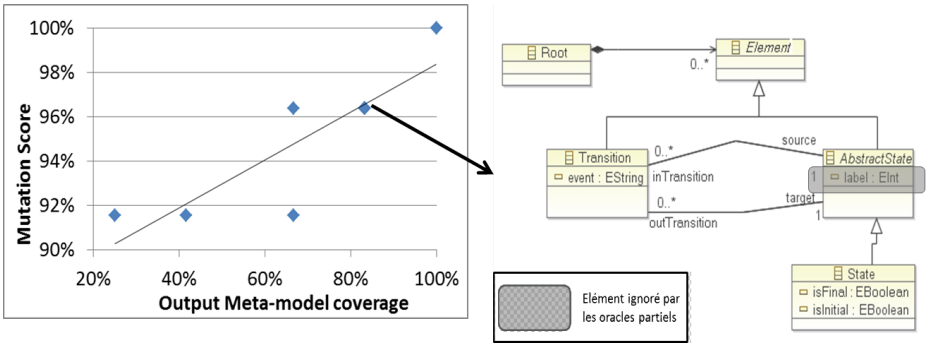


Figure B.6 – Lien entre score de mutation et taux de couverture pour la transformation fsm2ffsm (ensemble P6)

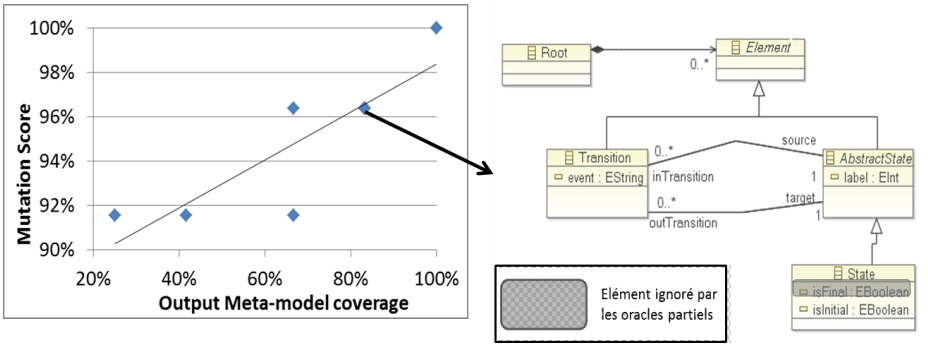


Figure B.7 – Lien entre score de mutation et taux de couverture pour la transformation fsm2ffsm (ensemble P7)

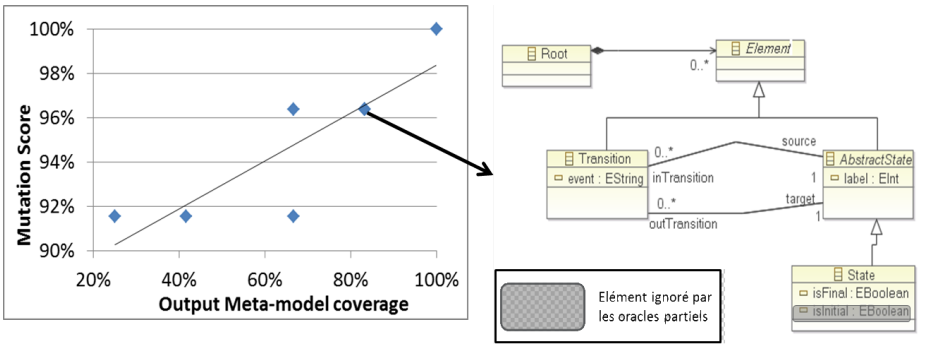


Figure B.8 – Lien entre score de mutation et taux de couverture pour la transformation fsm2ffsm (ensemble P8)

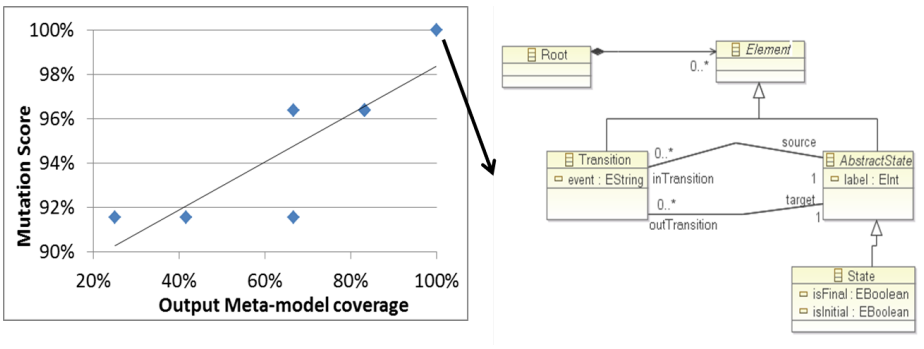


Figure B.9 – Lien entre score de mutation et taux de couverture pour la transformation fsm2ffsm (ensemble P0

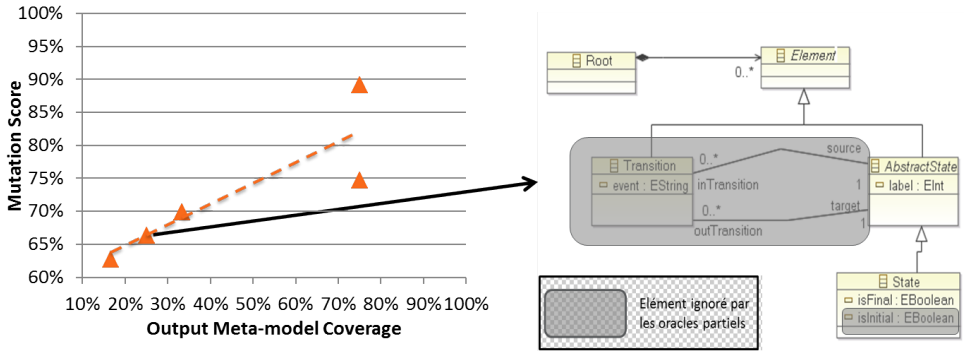


Figure B.10 – Lien entre score de mutation et taux de couverture pour la transformation fsm2ffsm (ensemble C2)

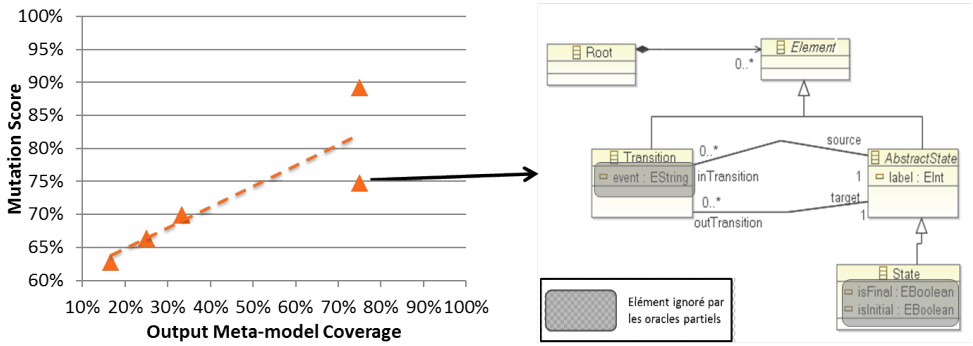


Figure B.11 – Lien entre score de mutation et taux de couverture pour la transformation fsm2ffsm (ensemble C4)

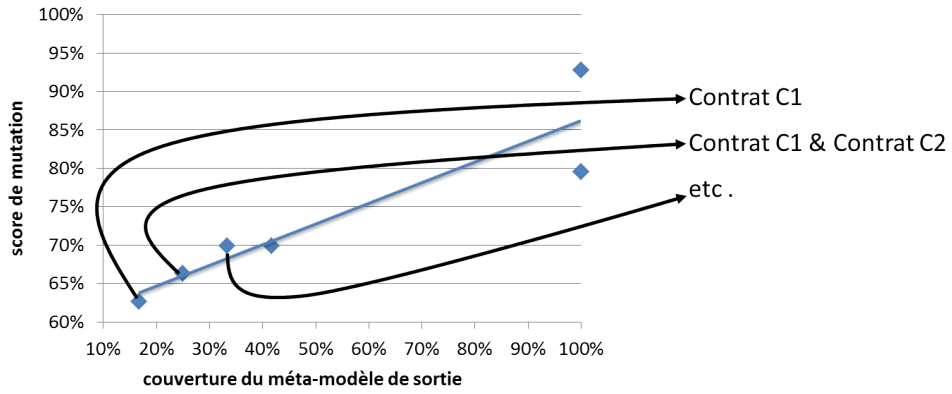
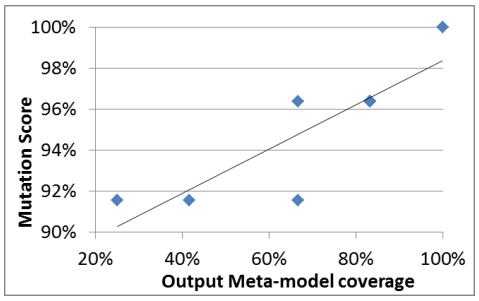
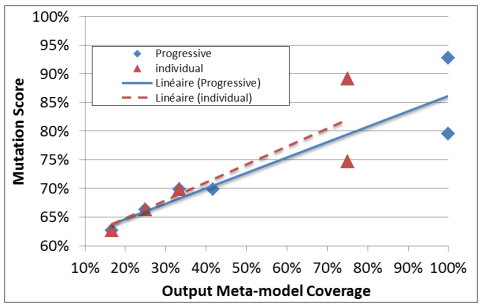


Figure B.12 – Lien entre score de mutation et taux de couverture pour la transformation fsm2ffsm (ensemble CIncr1)

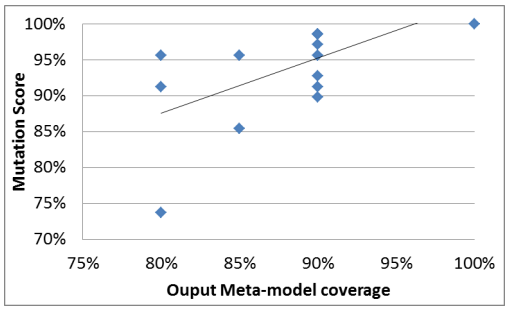


(a) Oracles utilisant des modèles de sortie attendus

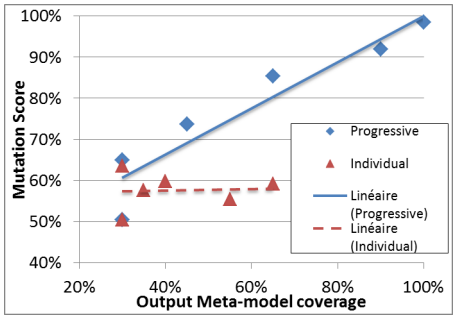


(b) Oracles utilisant des contrats

Figure B.13 – Lien entre score de mutation et taux de couverture pour la transformation fsm2ffsm



(a) Oracles utilisant des modèles de sortie attendus



(b) Oracles utilisant des contrats

Figure B.14 – Lien entre score de mutation et taux de couverture pour la transformation UML2CSP

Diagramme d'activité UML vers CSP

Nous créons seize ensembles d'oracles partiels pour la transformation d'un diagramme d'activité UML en un programme CSP :

1. les oracles de l'ensemble **Q1** ignorent toutes les instances de la méta-classe `Process` ;
2. les oracles de l'ensemble **Q2** ignorent les instances de la méta-classe `Prefix` ;
3. les oracles de l'ensemble **Q3** ignorent les instances de la méta-classe `Event` ;
4. les oracles de l'ensemble **Q4** ignorent les instances de la méta-classe `Concurrency` ;
5. les oracles de l'ensemble **Q5** ignorent les instances de la méta-classe `Condition` ;
6. les oracles de l'ensemble **Q6** ignorent les instances de la référence `processIdentifier` ;
7. les oracles de l'ensemble **Q7** ignorent les instances de la référence `process` ;
8. les oracles de l'ensemble **Q8** ignorent les instances de la référence `targetProcess` ;
9. les oracles de l'ensemble **Q9** ignorent les instances de la référence `event` ;
10. les oracles de l'ensemble **Q10** ignorent les instances de la référence `leftHandSide` ;
11. les oracles de l'ensemble **Q11** ignorent les instances de la référence `rightHandSide` ;
12. les oracles de l'ensemble **Q12** ignorent les instances de l'attribut `CSPContainer.name` ;
13. les oracles de l'ensemble **Q13** ignorent les instances de l'attribut `Process.name` ;
14. les oracles de l'ensemble **Q14** ignorent les instances de l'attribut `Event.name` ;
15. les oracles de l'ensemble **Q15** ignorent les instances de l'attribut `Condition.expression` ;
16. les oracles de l'ensemble **Q0** n'ignorent aucun élément.

Nous créons également six contrats individuels pour cette transformation :

1. le contrat **D1**, présenté Listing B.1, exprime que les instances d'`Action` ainsi que celles de `Edge` qui les entourent sont correctement transformées ;
2. le contrat **D2** exprime que les instances de `Edge` qui ciblent une instance de `FinalNode` sont correctement transformées (cf Listing B.2) ;
3. le contrat **D3** exprime que les instances d'`MergeNode` ainsi que celles de `Edge` qui les entourent sont correctement transformées (cf Listing B.3) ;
4. le contrat **D4** exprime que les instances d'`JoinNode` ainsi que celles de `Edge` qui les entourent sont correctement transformées (cf Listing B.4) ;
5. le contrat **D5** exprime que les instances d'`ForkNode` ainsi que celles de `Edge` qui les entourent sont correctement transformées (cf Listing B.5) ;

6. le contrat **D6** exprime que les instances d'`DecisionNode` ainsi que celles de `Edge` qui les entourent sont correctement transformées (cf Listing B.6).

```
//Checks that actions and the edges arround them are correctly transformed
operation validate1(act : ActivityDiagram, csp : CspContainer, resPath : String) :
    Boolean is do
        var res : Boolean

        res := act.edges.select{e | e.target.isInstanceOf(Action)}.forAll{e2 |
                csp.processAssignments.select{pa |
                        pa.processIdentifier.name == e2.name and
                        pa.process.isInstanceOf(Prefix)}.select{pa2 |
                                pa2.process.asType(Prefix).event.name == e2.target.
                                    asType(Action).name and
                                pa2.process.asType(Prefix).targetProcess.isInstanceOf(
                                    Process)}.exists{pa3 |
                                        act.edges.exists{e3 | e3.source == e2.target and
                                        pa3.process.asType(Prefix).targetProcess.asType(
                                            Process).name == 3.name}}
                }

        result := res
end
```

Listing B.1 – Le contrat D1

```
//Checks that edges targeting final states are correctly transformed
operation validate2(act : ActivityDiagram, csp : CspContainer, resPath : String) :
    Boolean is do
        var res : Boolean
        res := act.edges.select{e |
                e.target.isInstanceOf(FinalNode)}.forAll{e2 |
                        csp.processAssignments.select{pa |
                                pa.processIdentifier.name == e2.name and
                                pa.process.isInstanceOf(Process)}.exists{pa2 |
                                        pa2.process.asType(Process).name == "SKIP" }}

        result := res
end
```

Listing B.2 – Le contrat D2

```
//Checks that merge nodes and the edges arround them are correctly transformed
operation validate3(act : ActivityDiagram, csp : CspContainer, resPath : String) :
    Boolean is do
        var res : Boolean
        res := act.edges.select{e | e.target.isInstanceOf(MergeNode)}.forAll{e2 |
                csp.processAssignments.select{pa | pa.processIdentifier.name == e2.name
                    and
                        pa.process.isInstanceOf(Process)}.exists{pa2 |
                                act.edges.exists{e3 |
                                        e3.source == e2.target and
                                        pa2.process.asType(Process).name == e3.name}}}

        result := res
end
```

Listing B.3 – Le contrat D3

```
//Checks that join nodes and the edges arround them are correctly transformed
operation validate4(act : ActivityDiagram, csp : CspContainer, resPath : String) :
    Boolean is do
```

```
        var res : Boolean
        res := act.edges.select{e | e.target.isInstanceOf(JoinNode)}.forAll{e2 |
                csp.processAssignments.select{pa | pa.processIdentifier.name == e2.name
                    and pa.process.isInstanceOf(Prefix)}.select{pa2 |
                        pa2.process.asType(Prefix).targetProcess.isInstanceOf(Process)}.
                            exists{pa3 |
                                act.edges.exists{e3 | e3.source == e2.target and
                                ( pa3.process.asType(Prefix).targetProcess.asType(
                                    Process).name == e3.name or pa3.process.asType(
                                    Prefix).targetProcess.asType(Process).name == "SKIP"
                                    ) and
                                pa3.process.asType(Prefix).event.name == "processJoin"
                                    and
                                csp.processAssignments.select{pa4 |
                                        pa4.process.isInstanceOf(Prefix)}.select{pa5 |
                                                pa5.process.asType(Prefix).targetProcess
                                                    .isInstanceOf(Process)}.select{pa6 |
                                                        pa6.process.asType(Prefix).
                                                            targetProcess.asType(Process
                                                            ).name == e3.name}.size() ==
                                                             1}}}
        result := res
end
```

Listing B.4 – Le contrat D4

```
//Checks concurrency
operation validate5(act : ActivityDiagram, csp : CspContainer, resPath : String) :
    Boolean is do
        var res : Boolean

        edges := act.edges.select{e | e.source.isInstanceOf(ForkNode)}
        res := csp.processAssignments.select{pa |
                pa.process.isInstanceOf(Concurrency)}.forAll{pa2 |
                        sub_validate_5(pa2.process.asType(Concurrency)) } and edges.size
                             == 0

        result := res
end

operation sub_validate_5(conc : Concurrency) : Boolean is do
        var res : Boolean

        if not (conc.leftHandSide.isInstanceOf(Process)) and
                not (conc.leftHandSide.isInstanceOf(Process)) and
                not (conc.leftHandSide.isInstanceOf(Concurrency))       and
                not (conc.rightHandSide.isInstanceOf(Concurrency)) then

                        res := false
        else
                if conc.leftHandSide.isInstanceOf(Process) and conc.rightHandSide.
                    isInstanceOf(Process) then

                        res := edges.exists{e |
                                e.name == conc.leftHandSide.asType(Process).name}
                        and
                        edges.exists{e |
                                e.name == conc.rightHandSide.asType(Process).name}

                        edges := edges.reject{e |
                                e.name == conc.leftHandSide.asType(Process).name or
                                e.name == conc.rightHandSide.asType(Process).name}
                else
                        if conc.leftHandSide.isInstanceOf(Process) then
```

```
                                res := edges.exists{e |
                                        e.name == conc.leftHandSide.asType(Process).name
                                            } and
                                sub_validate_5(conc.rightHandSide.asType(Concurrency))

                                edges := edges.reject{e | e.name == conc.leftHandSide.
                                    asType(Process).name}
                        else
                                if conc.rightHandSide.isInstanceOf(Process) then

                                        res := edges.exists{e |
                                                e.name == conc.rightHandSide.asType(
                                                    Process).name} and
                                        sub_validate_5(conc.leftHandSide.asType(
                                            Concurrency))

                                        edges := edges.reject{e |
                                                e.name == conc.rightHandSide.asType(
                                                    Process).name}
                                else

                                        res := sub_validate_5(conc.leftHandSide.asType(
                                            Concurrency)) and
                                        sub_validate_5(conc.rightHandSide.asType(
                                            Concurrency))
                                end
                        end
                end
        end
        result := res
end
```

Listing B.5 – Le contrat D5

```
//Checks Condition
operation validate6(act : ActivityDiagram, csp : CspContainer, resPath : String) :
    Boolean is do

        var res : Boolean

        edges := act.edges.select{e |
                e.source.isInstanceOf(DecisionNode)}
        res := csp.processAssignments.select{pa |
                pa.process.isInstanceOf(Condition)}.forAll{pa2 |
                        sub_validate_6(pa2.process.asType(Condition)) } and
        edges.size == 0


        result := res

end

operation sub_validate_6(cond : Condition) : Boolean is do
        var res : Boolean

        if not cond.leftHandSide.isInstanceOf(Process) and
                not cond.rightHandSide.isInstanceOf(Process) and
                not cond.rightHandSide.isInstanceOf(Condition) then

                res := false
        else
                if not edges.exists{e |
                                e.name == cond.leftHandSide.asType(Process).name and
                        e.guard == cond.expression} then
```

```
                    res := false
                else
                    edges := edges.reject{e |
                            e.name == cond.leftHandSide.asType(Process).name and
                            e.guard == cond.expression}

                    if cond.rightHandSide.isInstanceOf(Process) then
                            res := edges.exists{e |
                                    e.name == cond.rightHandSide.asType(Process).
                                        name and
                                    e.guard == "else"}

                            edges := edges.reject{e |
                                    e.name == cond.rightHandSide.asType(Process).
                                        name and
                                    e.guard == "else"}
                    else
                            res := sub_validate_6(cond.rightHandSide.asType(
                                Condition))
                    end
                end
        end

        result := res
end
```

Listing B.6 – Le contrat D6

Bibliographie

[Amar et al., 2008] Amar, B., Leblanc, H., and Coulette, B. (2008). A Traceability Engine Dedicated to Model Transformation for Software Engineering. In *ECMDA Traceability Workshop*, pages 08–12.

[Ammann and Offutt, 2008] Ammann, P. and Offutt, J. (2008). *Introduction to Software Testing*. Cambridge University Press.

[Amrani et al., 2012] Amrani, M., Lucio, L., Selim, G. M. K., Combemale, B., Dingel, J., Vangheluwe, H., Traon, Y. L., and Cordy, J. R. (2012). A Tridimensional Approach for Studying the Formal Verification of Model Transformations. In Antoniol, G., Bertolino, A., and Labiche, Y., editors, *ICST*, pages 921–928. IEEE.

[Andrews et al., 2003] Andrews, A., France, R., Ghosh, S., and Craig, G. (2003). Test adequacy criteria for UML design models. *Software Testing, Verification and Reliability*, 13(2) :95–127.

[Aranega et al., 2008] Aranega, V., Mottu, J.-m., Etien, A., and Dekeyser, J.-l. (2008). Traceability Mechanism for Error Localization in Model Transformation. *Test*.

[Aranega et al., 2011a] Aranega, V., Mottu, J.-M., Etien, A., and Dekeyser, J.-L. (2011a). Traceability for Mutation Analysis in Model Transformation. In Dingel, J. and Solberg, A., editors, *Models in Software Engineering*, volume 6627 of *Lecture Notes in Computer Science*, pages 259–273. Springer Berlin / Heidelberg.

[Aranega et al., 2011b] Aranega, V., Mottu, J.-M., Etien, A., and Dekeyser, J.-L. (2011b). Traceability for Mutation Analysis in Model Transformation. In Dingel, J. and Solberg, A., editors, *Models in Software Engineering*, volume 6627 of *Lecture Notes in Computer Science*, pages 259–273. Springer Berlin / Heidelberg.

[Attarha and Modiri, 2011] Attarha, M. and Modiri, N. (2011). Focusing on the Importance and the Role of Requirement Engineering. In *Interaction Sciences (ICIS), 2011 4th International Conference on*, pages 181–184. IEEE.

[Baresi and Young, 2001] Baresi, L. and Young, M. (2001). Test Oracles. Technical report.

[Baudry et al., 2011] Baudry, B., Bazex, P., Dalbin, J.-C., Dhaussy, P., Dubois, H., Percebois, C., Poupart, E., and Sabatier, L. (2011). Trust in MDE Components : the DOMINO Experiment (regular paper). In *International Workshop on Security and Dependability for Resource Constrained Embedded Systems, Vienne, Autriche, 14/09/2010*, page (on line), http ://portal.acm.org/dl.cfm. ACM DL.

[Baudry et al., 2010] Baudry, B., Ghosh, S., Fleurey, F., France, R., Le Traon, Y., and Mottu, J.-M. (2010). Barriers to Systematic Model Transformation Testing. *Communications of the ACM*, 53(6).

[Bauer and Küster, 2011] Bauer, E. and Küster, J. (2011). Combining Specification-Based and Code-Based Coverage for Model Transformation Chains. *Theory and Practice of Model Transformations*, pages 78–92.

[Beizer, 1990] Beizer, B. (1990). *Software Testing Techniques (2. ed.)*. Van Nostrand Reinhold.

[Bertolino, 2007] Bertolino, A. (2007). Software Testing Research : Achievements, Challenges, Dreams. In Briand, L. C. and Wolf, A. L., editors, *FOSE*, pages 85–103.

[Beugnard et al., 1999] Beugnard, A., Jezéquel, J.-M., Plouzeau, N., and Watkins, D. (1999). Making Components Contract Aware. *Computer*, 32(7) :38 – 45.

[Bézivin, 2004] Bézivin, J. (2004). In search of a Basic Principle for Model-Driven Engineering. *Novatica – Special Issue on UML (Unified Modeling Language)*, 5(2) :21–24.

[Bisztray et al., 2007] Bisztray, D., Ehrig, K., and Heckel, R. (2007). Case study : UML to CSP transformation. *AGTIVE*.

[Braga et al., 2011] Braga, C., Menezes, R., Comicio, T., Santos, C., and Landim, E. (2011). On the Specification, Verification and Implementation of Model Transformations with Transformation Contracts. In *SBMF*.

[Brun and Pierantonio, 2008] Brun, C. and Pierantonio, A. (2008). Model Differences in the Eclipse Modelling Framework. *UPGRADE, The European J. for the Informatics Professional*.

[Büttner et al., 2011] Büttner, F., Cabot, J., and Gogolla, M. (2011). On Validation of ATL Transformation Rules by Transformation Models. In *MoDeVVa*.

[Cabot et al., 2010] Cabot, J., Clarisó, R., Guerra, E., and De Lara, J. (2010). Verification and Validation of Declarative Model-to-Model Transformations through Invariants. *JSS*, 83.

[Cariou et al., 2009] Cariou, E., Belloir, N., Barbier, F., and Djemam, N. (2009). OCL Contracts for the Verification of Model Transformations. In *the Workshop The Pragmatics of OCL and Other Textual Specification Languages at MoDELS 2009*, volume 24. Electronic Communications of the EASST.

[Chen and Avizienis, 1995] Chen, L. and Avizienis, A. (1995). N-Version Programming : a Fault-Tolerance Approach to Reliability of Software Operation. In *Fault-Tolerant Computing, 1995, Highlights from Twenty-Five Years., Twenty-Fifth International Symposium on*, pages 113–.

[Cicchetti et al., 2008] Cicchetti, A., Di Ruscio, D., Eramo, R., and Pierantonio, A. (2008). Meta-model Differences for Supporting Model Co-evolution. In *proceedings of the 2nd Workshop on Model-Driven Software Evolution, MoDSE'2008*.

[Cicchetti et al., 2007] Cicchetti, A., Di Ruscio, D., and Pierantonio, A. (2007). A Metamodel Independent Approach to Difference Representation. *Technology*, 6(9) :165–185.

[Czarnecki et al., 2009] Czarnecki, K., Foster, J., Hu, Z., Lämmel, R., Schürr, A., and Terwilliger, J. (2009). Bidirectional Transformations : A Cross-Discipline Perspective. In Paige, R., editor, *Theory and Practice of Model Transformations*, volume 5563 of *Lecture Notes in Computer Science*, pages 260–283. Springer Berlin Heidelberg.

[Dalal et al., 1999] Dalal, S., Jain, A., Karunanithi, N., Leaton, J., Lott, C., Patton, G., and Horowitz, B. (1999). Model-Based Testing in Practice. In *ICSE*, pages 285 –294.

[de O. Braga et al., 2012] de O. Braga, C., Menezes, R., Comicio, T., Santos, C., and Landim, E. (2012). Transformation Contracts in Practice. *IET Software*, 6(1) :16–32.

[DeMillo et al., 1978] DeMillo, R. A., Lipton, R. J., and Sayward, F. G. (1978). Hints on Test Data Selection : Help for the Practicing Programmer. *Computer*, 11 :34–41.

[Dijkstra, 1972] Dijkstra, E. W. (1972). The Humble Programmer. *Commun. ACM*, 15(10) :859–866.

[Ehrig et al., 2010a] Ehrig, H., Habel, A., Lambers, L., Orejas, F., and Golas, U. (2010a). Local Confluence for Rules with Nested Application Conditions. In [Ehrig et al., 2010b], pages 330–345.

[Ehrig et al., 2010b] Ehrig, H., Rensink, A., Rozenberg, G., and Schürr, A., editors (2010b). *Graph Transformations - 5th International Conference, ICGT 2010, Enschede, The Netherlands, September 27 - - October 2, 2010. Proceedings*, volume 6372 of *Lecture Notes in Computer Science*. Springer.

[Finot et al.,] Finot, O., Mottu, J.-M., Sunyé, G., and Attiogbe, C. Matériel expérimental. http://pagesperso.lina.univ-nantes.fr/%7Emottu-jm/development-en.html.

[Finot et al., 2012a] Finot, O., Mottu, J.-M., Sunye, G., and Attiogbe, C. (2012a). Comparaison de modèles filtrée pour le test de transformations de modèles. In *Conférence en IngénieriE du Logiciel CIEL 2012*, Rennes, France.

[Finot et al., 2012b] Finot, O., Mottu, J.-M., Sunye, G., and Attiogbe, C. (2012b). Comparaison de modèles filtrée pour le test de transformations de modèles. Poster présenté aux journées GDR-GPL 2012, Rennes, France.

[Finot et al., 2012c] Finot, O., Mottu, J.-M., Sunye, G., and Attiogbe, C. (2012c). Filtered Comparison for Oracle in ModelTransformation Testing. In *Proceedings of the ICTSS 2012 Ph.D. Workshop*, pages 13–17, Aalborg, Denmark.

[Finot et al., 2013a] Finot, O., Mottu, J.-M., Sunye, G., and Attiogbe, C. (2013a). Partial test oracle in model transformation testing. In *International Conference on Model Transformation*, Budapest, Hungary.

[Finot et al., 2013b] Finot, O., Mottu, J.-M., Sunye, G., and Degueule, T. (2013b). Using meta-model coverage to qualify test oracles. In *Analysis of Model Transformations*, Miami, Florida, USA.

[Fleurey et al., 2009] Fleurey, F., Baudry, B., Muller, P.-A., and Traon, Y. L. (2009). Qualifying Input Test Data for Model Transformations. *Software and System Modeling*, 8(2) :185–203.

[Förtsch and Westfechtel, 2007] Förtsch, S. and Westfechtel, B. (2007). Differencing and Merging of Software Diagrams - State of the Art and Challenges. In Filipe, J., Helfert, M., and Shishkov, B., editors, *Proceedings of the Second International Conference on Software and Data Technologies (ICSOFT 2007)*, pages 90–99, Barcelona, Spain. INSTICC Press, Setubal, Portugal.

[Fraser and Zeller, 2010] Fraser, G. and Zeller, A. (2010). Mutation-driven Generation of Unit Tests and Oracles. In *ISSTA*, pages 147–158.

[García-Domínguez et al., 2011] García-Domínguez, A., Kolovos, D., Rose, L., Paige, R., and Medina-Bulo, I. (2011). EUnit : a Unit Testing Framework for Model Management Tasks. *MoDELS*.

[Giese et al., 2006] Giese, H., Glesner, S., Leitner, J., Schäfer, W., and Wagner, R. (2006). Towards Verified Model Transformations. *MoDeV²a*.

[Gogolla and Vallecillo, 2011] Gogolla, M. and Vallecillo, A. (2011). Tractable Model Transformation Testing. *ECMFA*.

[González and Cabot, 2012] González, C. A. and Cabot, J. (2012). ATLTest : A White-Box Test Generation Approach for ATL Transformations. In *MoDELS*, pages 449–464.

[Guerra, 2012] Guerra, E. (2012). Specification-Driven Test Generation for Model Transformations. In *ICMT*, pages 40–55.

[Guerra et al., 2013] Guerra, E., Lara, J., Wimmer, M., Kappel, G., Kusel, A., Retschitzegger, W., Schonbock, J., and Schwinger, W. (2013). Automated Verification of Model Transformations Based on Visual Contracts. *Automated Software Engineering*, 20(1) :5–46.

[Heimdahl et al., 2004] Heimdahl, M. P. E., Devaraj, G., and Weber, R. (2004). Specification Test Coverage Adequacy Criteria = Specification Test Generation Inadequacy Criteria ? In *HASE*, pages 178–186. IEEE Computer Society.

[Hermann et al., 2010] Hermann, F., Ehrig, H., Orejas, F., and Golas, U. (2010). Formal Analysis of Functional Behaviour for Model Transformations Based on Triple Graph Grammars. In [Ehrig et al., 2010b], pages 155–170.

[Hoare, 1978] Hoare, C. A. R. (1978). Communicating Sequential Processes.

[Hoffman, 1998] Hoffman, D. (1998). A Taxonomy for Test Oracles. 98 :52–60.

[Holt et al., 2009] Holt, N., Arisholm, E., and Briand, L. (2009). An Eclipse Plug-in for the Flattening of Concurrency and Hierarchy in UML State Machines. Technical report.

[Hsia et al., 1994] Hsia, P., Samuel, J., Gao, J., Kung, D. C., Toyoshima, Y., and Chen, C. (1994). Formal Approach to Scenario Analysis. *IEEE Software*, 11(2) :33–41.

[Jeanneret et al., 2011] Jeanneret, C., Glinz, M., and Baudry, B. (2011). Estimating Footprints of Model Operations. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 601 – 610, Honolulu, USA. IEEE.

[Jézéquel et al., 2001] Jézéquel, J.-M., Deveaux, D., and Le Traon, Y. (2001). Reliable Objects : a Lightweight Approach Applied to Java. *IEEE Software*, 18(4) :76–83.

[Jouault, 2005] Jouault, F. (2005). Loosely Coupled Traceability for ATL. In *In Proceedings of the European Conference on Model Driven Architecture (ECMDA) workshop on traceability*, pages 29–37.

[Jouault et al., 2008] Jouault, F., Allilaire, F., Bézivin, J., and Kurtev, I. (2008). ATL : A Model Transformation Tool. *Science of Computer Programming*, 72 :31 – 39. Special Issue on Second issue of experimental software and toolkits (EST).

[Kent, 2002] Kent, S. (2002). Model Driven Engineering. In Butler, M. J., Petre, L., and Sere, K., editors, *IFM*, volume 2335 of *Lecture Notes in Computer Science*, pages 286–298. Springer.

[Kessentini et al., 2011] Kessentini, M., Sahraoui, H. A., and Boukadoum, M. (2011). Example-Based Model-transformation Testing. *Autom. Softw. Eng.*, 18(2) :199–224.

[Khan and Hassine, 2013] Khan, Y. and Hassine, J. (2013). Mutation Operators for the Atlas Transformation Language. In *Mutation*.

[Kleppe et al., 2003] Kleppe, A. G., Warmer, J., and Bast, W. (2003). *MDA Explained : The Model Driven Architecture : Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[Knauth et al., 2009] Knauth, T., Fetzer, C., and Felber, P. (2009). Assertion-Driven Development : Assessing the Quality of Contracts Using Meta-Mutations. In *ICST Workshops*, pages 182–191.

[Kolovos et al., 2006] Kolovos, D., Paige, R., and Polack, F. (2006). Model Comparison : a Foundation for Model Composition and Model Transformation Testing. In *Proceedings of the 2006 international workshop on Global integrated model management*, pages 13–20. ACM.

[Kolovos et al., 2009] Kolovos, D. S., Di Ruscio, D., Pierantonio, A., and Paige, R. F. (2009). Different Models for Model Matching : An Analysis of Approaches to Support Model Differencing. In *Proceedings of the 2009 ICSE Workshop on Comparison and Versioning of Software Models*, CVSM '09, pages 1–6, Washington, DC, USA. IEEE Computer Society.

[Küster, 2006] Küster, J. M. (2006). Definition and Validation of Model Transformations. *Software and System Modeling*, 5(3) :233–259.

[Küster and Abd-El-Razik, 2006] Küster, J. M. and Abd-El-Razik, M. (2006). Validation of Model Transformations - First Experiences Using a White Box Approach. In Kühne, T., editor, *MoDELS Workshops*, volume 4364 of *Lecture Notes in Computer Science*, pages 193–204. Springer.

[Lin et al., 2007] Lin, Y., Gray, J., and Jouault, F. (2007). DSMDiff : a Differentiation Tool for Domain-specific Models. *European Journal of Information Systems*, 16(4) :349–361.

[Lin et al., 2005] Lin, Y., Zhang, J., and Gray, J. (2005). A Testing Framework for Model Transformations. *Model-driven software development*, pages 219–236.

[Lopes et al., 2005] Lopes, D., Hammoudi, S., Bézivin, J., and Jouault, F. (2005). Generating Transformation Definition from Mapping Specification : Application to Web Service Platform. In Pastor, O. and e Cunha, J. F., editors, *CAiSE*, volume 3520 of *Lecture Notes in Computer Science*, pages 309–325. Springer.

[Lopes et al., 2006] Lopes, D., Hammoudi, S., de Souza, J., and Bontempo, A. (2006). Metamodel Matching : Experiments and Comparison. In *ICSEA*, page 2. IEEE Computer Society.

[Lúcio et al., 2010] Lúcio, L., Barroca, B., and Amaral, V. (2010). A Technique for Automatic Validation of Model Transformations. *MoDELS*.

[Manolache and Kourie, 2001] Manolache, L. I. and Kourie, D. G. (2001). Software Testing Using Model Programs. *Software : Practice and Experience*, 31(13) :1211–1236.

[McQuillan and Power, 2009] McQuillan, J. A. and Power, J. F. (2009). White-box Coverage Criteria for Model Transformations. *Model Transformation with ATL*, page 63.

[Mehra et al., 2005] Mehra, A., Grundy, J., and Hosking, J. (2005). A Generic Approach to Supporting Diagram Differencing and Merging for Collaborative Design. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, ASE '05, pages 204–213, New York, NY, USA. ACM.

[Mens, 2002] Mens, T. (2002). A State-of-the-Art Survey on Software Merging. *IEEE Trans. Softw. Eng.*, 28 :449–462.

[Meyer, 1992] Meyer, B. (1992). Applying 'Design by Contract'. *Computer*, 25(10) :40 –51.

[Mottu, 2008] Mottu, J. (2008). *Oracles et qualification du test de transformations de modèles*. PhD thesis, Université de Rennes 1.

[Mottu et al., 2006a] Mottu, J.-M., Baudry, B., and Le Traon, Y. (2006a). Mutation Analysis Testing for Model Transformations. In *proceedings of the European Conference on Model Driven Architecture (ECMDA 06)*, Bilbao, Spain.

[Mottu et al., 2006b] Mottu, J.-M., Baudry, B., and Le Traon, Y. (2006b). Reusable MDA Components : A Testing-for-Trust Approach. In *proceedings of the MoDELS/UML 2006*, Genova, Italy.

[Mottu et al., 2008] Mottu, J.-M., Baudry, B., and Le Traon, Y. (2008). Model Transformation Testing : Oracle Issue. In *MoDeVVa'08*.

[Muller et al., 2005] Muller, P.-A., Fleurey, F., and Jézéquel, J.-M. (2005). Weaving Executability into Object-Oriented Meta-languages. In Briand, L. C. and Williams, C., editors, *MoDELS*, volume 3713 of *Lecture Notes in Computer Science*, pages 264–278. Springer.

[Murphy et al., 2009] Murphy, C., Shen, K., and Kaiser, G. E. (2009). Automatic System Testing of Programs Without Test Oracles. In [Rothermel and Dillon, 2009], pages 189–200.

[Nagappan et al., 2005] Nagappan, N., Williams, L., Osborne, J., Vouk, M., and Abrahamsson, P. (2005). Providing Test Quality Feedback Using Static Source Code and Automatic Test Suite Metrics. In *Software Reliability Engineering, 2005. ISSRE 2005. 16th IEEE International Symposium on*, pages 10 pp.–94.

[Narayanan and Karsai, 2008a] Narayanan, A. and Karsai, G. (2008a). Specifying the Correctness Properties of Model Transformations. GRaMoT '08.

[Narayanan and Karsai, 2008b] Narayanan, A. and Karsai, G. (2008b). Verifying Model Transformations by Structural Correspondence. *Communications of the EASST*, 10(0).

[Ostrand and Balcer, 1988] Ostrand, T. J. and Balcer, M. J. (1988). The Category-partition Method for Specifying and Generating Fuctional Tests. *Commun. ACM*, 31(6) :676–686.

[Polikarpova et al., 2009] Polikarpova, N., Ciupa, I., and Meyer, B. (2009). A Comparative Study of Programmer-written and Automatically Inferred Contracts. In [Rothermel and Dillon, 2009], pages 93–104.

[Pretschner et al., 2005] Pretschner, A., Prenninger, W., Wagner, S., Kühnel, C., Baumgartner, M., Sostawa, B., Zölch, R., and Stauner, T. (2005). One Evaluation of Model-Based Testing and its Automation. In Roman, G.-C., Griswold, W. G., and Nuseibeh, B., editors, *ICSE*, pages 392–401. ACM.

[Ramos et al., 2007] Ramos, R., Barais, O., and Jézéquel, J.-M. (2007). Matching model-snippets. In *In Proceedings of ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems (MoDELS 07)*, Nashville, TN, USA.

[Regnell et al., 1995] Regnell, B., Kimbler, K., and Wesslén, A. (1995). Improving the Use Case Driven Approach to Requirements Engineering. In *RE*, pages 40–41. IEEE Computer Society.

[Reichhart et al., 2007] Reichhart, S., G ?ırba, T., and Ducasse, S. (2007). Rule-Based Assessment of Test Quality. *JOURNAL OF OBJECT TECHNOLOGY*, 6(9).

[Rosenberg et al., 1998] Rosenberg, L. H., Theodore, P., Hammer, F., and Huffman, L. L. (1998). Requirements, Testing and Metrics. In *In 15th Annual Pacific Northwest Software Quality Conference*.

[Rothermel and Dillon, 2009] Rothermel, G. and Dillon, L. K., editors (2009). *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, ISSTA 2009, Chicago, IL, USA, July 19-23, 2009*. ACM.

[Schätz, 2010] Schätz, B. (2010). Verification of Model Transformations. *EASST*.

[Schuler and Zeller, 2011] Schuler, D. and Zeller, A. (2011). Assessing Oracle Quality with Checked Coverage. In *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on*, pages 90–99.

[Selim et al., 2012] Selim, G. M. K., Cordy, J. R., and Dingel, J. (2012). Model Transformation Testing : the State of the Art. In *Proceedings of the First Workshop on the Analysis of Model Transformations*, AMT '12, pages 21–26, New York, NY, USA. ACM.

[Sen et al., 2009] Sen, S., Moha, N., Baudry, B., and Jézéquel, J. (2009). Meta-model Pruning. *Model Driven Engineering Languages and Systems*, 215483 :32–46.

[Sen et al., 2012] Sen, S., Mottu, J.-M., Tisi, M., and Cabot, J. (2012). Using Models of Partial Knowledge to Test Model Transformations. In *International Conference on Model Transformation*, Prague, Czech Republic.

[Staats et al., 2012] Staats, M., Gay, G., and Heimdahl, M. P. E. (2012). Automated Oracle Creation Support, or : How I learned to Stop Worrying About Fault Propagation and Love Mutation Testing. In *ICSE*, pages 870–880.

[Staats et al., 2011] Staats, M., Whalen, M. W., and Heimdahl, M. P. E. (2011). Programs, Tests, and Oracles : the Foundations of Testing Revisited. In *ICSE*, pages 391–400.

[Stevens, 2008] Stevens, P. (2008). A Landscape of Bidirectional Model Transformations. In Lämmel, R., Visser, J., and Saraiva, J., editors, *Generative and Transformational Techniques in Software Engineering II*, volume 5235 of *Lecture Notes in Computer Science*, pages 408–424. Springer Berlin Heidelberg.

[Tiso et al., 2012] Tiso, A., Reggio, G., and Leotta, M. (2012). Early Experiences on Model Transformation Testing. In *Proceedings of 1st Workshop on the Analysis of Model Transformations (AMT 2012 co-located with MoDELS 2012)*.

[Tretmans, 2008] Tretmans, J. (2008). Model Based Testing with Labelled Transition Systems. In Hierons, R. M., Bowen, J. P., and Harman, M., editors, *Formal Methods and Testing*, volume 4949 of *Lecture Notes in Computer Science*, pages 1–38. Springer.

[Utting et al., 2012] Utting, M., Pretschner, A., and Legeard, B. (2012). A Taxonomy of Model-Based Testing Approaches. *Softw. Test., Verif. Reliab.*, 22(5) :297–312.

[Vallecillo et al., 2012] Vallecillo, A., Gogolla, M., Burgueño, L., Wimmer, M., and Hamann, L. (2012). Formal Specification and Testing of Model Transformations. In *SFM*, pages 399–437.

[Weyuker, 1982] Weyuker, E. J. (1982). On Testing Non-Testable Programs. *Comput. J.*, 25(4) :465–470.

[Winkler and Pilgrim, 2010] Winkler, S. and Pilgrim, J. (2010). A Survey of Traceability in Requirements Engineering and Model-driven Development. *Softw. Syst. Model.*, 9(4) :529–565.

[Wu et al., 2012] Wu, H., Monahan, R., and Power, J. (2012). Metamodel Instance Generation : A Systematic Literature Review. *arXiv preprint arXiv :1211.6322*, pages 1–24.

[Xing and Stroulia, 2005] Xing, Z. and Stroulia, E. (2005). UMLDiff : an Algorithm for Object-oriented Design Differencing. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, ASE '05, pages 54–65, New York, NY, USA. ACM.

Liste des figures

Résumé

L'Ingénierie Dirigée par les Modèles place les modèles au cœur du cycle de développement logiciel. Ces modèles évoluent par le biais de diverses transformations. Dans cette thèse nous nous sommes intéressés à la validation de ces transformations de modèles par le test, et en particulier à l'oracle de ce test. Nous proposons deux approches pour assister le testeur dans la création de ces oracles. Tout d'abord, nous offrons une assistance passive en fournissant au testeur une nouvelle fonction d'oracle. Cette dernière lui permet de créer des oracles qui ne contrôlent qu'une partie des modèles obtenus. Nous avons défini la notion de verdict partiel, explicité les situations où un verdict partiel est plus avantageux et proposé un protocole global du test de transformations dans ce contexte. Nous avons mis en œuvre cette première proposition dans un outillage avec lequel nous l'avons expérimentée. Ensuite, nous offrons au testeur une assistance active en étudiant la qualité d'un ensemble d'oracles. Nous considérons la qualité d'un ensemble d'oracles selon sa capacité à détecter des fautes dans la transformation sous test. Nous proposons une méthode qui corrige en partie les insuffisances de l'analyse de mutation, utilisée dans ce contexte ; nous mesurons la couverture du méta-modèle de sortie par l'ensemble d'oracles considéré. Nous montrons que notre approche est indépendante du langage utilisé pour la mise en œuvre de la transformation sous test, et fournit au testeur des informations pour l'amélioration des oracles. Nous avons défini une démarche pour mesurer la couverture et qualifier des oracles. Nous avons développé un outil pour expérimenter et valider notre proposition.

Mots-clés : Ingénierie Dirigée par les Modèles, Transformations de modèles, Oracle, Qualification

Abstract

With Model Driven Engineering models are the heart of software development. These models evolve through transformations. In this thesis our interest was the validation for these model transformations by testing, and more precisely the test oracles. We propose two approaches to assist the tester to create these oracles. With the first approach this assistance is passive; we provide the tester with a new oracle function. The test oracles created with this new oracle function control only part of the model produced by the transformation under test. We defined the notion of partial verdict, described the situations where having a partial verdict is beneficial for the tester and how to test a transformation in this context. We developed a tool implementing this proposal, and ran experiments with it. With the second approach, we provide a more active assistance about test oracles' quality. We study the quality of a set of model transformation test oracles. We consider that the quality of a set of oracles is linked to its ability to detect faults in the transformation under test. We show the limits of mutation analysis which is used for this purpose, then we propose a new approach that corrects part of these drawbacks. We measure the coverage of the output meta-model by the set of oracles we consider. Our approach does not depend on the language used for the transformation under test's implementation. It also provides the tester with hints on how to improve her oracles. We defined a process to evaluate meta-model coverage and qualify test oracles. We developed a tool implementing our approach to validate it through experimentations.

Keywords: Model Driven Engineering, Model Transformations, Oracle, Qualification